Achieving Security and Privacy via Encrypted Architectures

by

Lauren M. Biernacki

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Professor Todd Austin, Chair
Professor Reetuparna Das
Assistant Professor Jean-Baptiste Jeannin
Assistant Professor Baris Kasikci

Lauren M. Biernacki
lbiernac@umich.edu
ORCID iD: 0000-0001-8511-2287

© Lauren M. Biernacki 2023

In loving memory of my grandparents—

Irene Biernacki *(1936-2020)* and Francis Biernacki *(1931-2008)*

Julienne Provost *(1924-2020)* and Armand Provost *(1924-2019)*

ACKNOWLEDGMENTS

Throughout this dissertation, I frequently use the pronoun "we." This usage acknowledges all of my research collaborators and mentors, which I could not have done this work without. To my Ph.D. advisor, Prof. Todd Austin, thank you for your support and encouragement. It is your unwavering enthusiasm and excitement that sparked my interest in security research and propelled me throughout this journey—especially in the face of many paper rejections. I am so grateful for the opportunity to have you as a mentor and to continue working with you as a colleague. I look forward to the many out-of-the-box ideas to come. To the rest of the Austin Research Group, thank you for all your feedback, and support throughout this journey. You have made the Beyster building feel like home. Thank you to Mark Gallagher for the sleepless nights working toward our first conference deadlines and for indulging in creating *Morpheus The Movie*. Thank you to Tarunesh Verma, Shibo Chen, Donayam Benti, and Yonathan Fisseha for our entertaining conversations about research and beyond. Finally, thank you to Meron Demissie, Kidus Workneh, and Biniyam Tiruye for inviting me to work with you early in your Ph.D. journies. It has been a pleasure to watch you grow as researchers, and you have impressed me endlessly.

To my dissertation committee—Prof. Reetuparna Das, Prof. Baris Kasikci, and Prof. Jean-Baptiste Jeannin—thank you for your research guidance, both as committee members and collaborators. Reetu and Jean-Baptiste, as well as Prof. Westley Weimer, thank you for your mentorship during my academic job search. I valued your advice greatly. Thank you to Prof. Sharad Malik for your mentorship, especially for your diligent guidance on [1] and [2]. You have made me a better writer and a more thoughtful researcher.

Thank you to the funding organizations that have supported my research. My early work was supported by DARPA under Contract HR0011-18-C-0019 as part of the System Security Integration Through Hardware and Firmware (SSITH) program. Later, my research was supported by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation (SRC) program co-sponsored by DARPA. Within these centers, I have had the opportunity to develop as a researcher through numerous meetings and symposia involving academic and industry leaders. These centers gave me the opportunity to make connections and discuss my research much more broadly than otherwise

possible, and I am very grateful to have been a part of them. Finally, I would like to thank the Computer Science and Engineering department for supporting my teaching experiences as a graduate student instructor and GEO #3550 for advocating for graduate students at the University of Michigan.

In addition to acknowledging my research colleagues and funding sources, I must also thank my teachers. When entering college, I knew little about computer science or graduate school. When entering graduate school, I knew little about research or hardware security. All of this was gifted to me through my teachers. To you, I am eternally grateful. Thank you to Prof. Keith Barker (KB) for demonstrating what it means to be a compassionate and excellent educator. You have inspired my teaching career and my work in computer architecture. Someday I will be truly honored for my students to refer to me as LB. To Audra Baleisis, Michigan's Center for Research on Learning and Teaching (CRLT), and my fellow Engineering Teaching Consultants (ETCs), thank you for teaching me to be a better educator. I have learned many valuable lessons from our thoughtful conversations that I will take forth in my career. Thank you to those who encouraged me to consider graduate school, especially Prof. Alexander Schwarzmann, Prof. Alexander Russell, and Prof. Laurent Michel. I simply would not be here without you.

Finally, I would like to acknowledge all of those who have supported me throughout my graduate school journey. To my parents, for teaching me perseverance and always making me feel capable despite doubt. In so many ways, I see you in myself. I am so grateful for all of the things you have taught me, including the lessons I have not yet realized. Thank you to my brother, Ryan, and sister-in-law, Shawna, for your support and for keeping my parents company while I have been away. I am so proud of the life you have built together. Thank you to all of my friends who welcomed me to Michigan and ultimately made it my home. To Anna, for finding the courage to reach out on Facebook Messenger. Our friendship has brought me such joy and because of you I look forward to every holiday. To Adam and Thomas, for the laughs and late nights that made graduate school pass by almost too quickly. Finally, to Connor for letting me move halfway across the country and then joining me here. I cannot put into words how thankful I am for your support in this journey, and I am so lucky to have taken it with you. I cannot wait to see where this adventure takes us next.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF FIGURES x

LIST OF TABLES xv

ABSTRACT xix

CHAPTER

1 Introduction 1

1.1 How Software Exploits are Constructed 2
1.2 The Role of Vulnerabilities and Information Assets 4
1.3 Toward Durable Defenses 5
1.4 Achieving Security and Privacy via Encrypted Architectures 7
1.5 Dissertation Contributions and Organization 9

2 Background: Vulnerability Sources, Security Attacks, and Durable Hardware Defenses 13

2.1 Introduction 14
2.1.1 Chapter Contributions 15
2.2 Modeling the Program Life Cycle 16
2.2.1 High-Level Programming 18
2.2.2 Compilation 20
2.2.3 Hardware Realization 22
2.2.4 Relation to Previous Works 25
2.3 Security Exploits 26
2.3.1 Vulnerabilities 26
2.3.2 Implementation Information Assets 30
2.3.3 Summary of Security Exploits 34
2.4 Classification of Defenses 35
2.4.1 Avoidance of Vulnerabilities 36
2.4.2 Enforcement of Program Semantics 37
2.4.3 Obfuscation of Implementation Information Assets 38
2.4.4 Summary of Defenses 39

2.5 Case Studies of Attack Genealogies . . . 40
2.5.1 Control-Flow Attacks . . . 40
2.5.2 Derandomization Attacks . . . 45
2.5.3 Timing Side-Channel Attacks . . . 48
2.5.4 Transient Execution Attacks . . . 54
2.6 Discussion and Limitations . . . 57
2.6.1 Defense Design Trends . . . 58
2.6.2 Performance-Security Trade-Offs . . . 59
2.6.3 Suggestions for Future Defenses . . . 60
2.7 Chapter Summary . . . 61
2.8 Implications for This Dissertation . . . 62

3 Architectural Approaches to Protecting Code and Pointers . . . 63

3.1 Introduction . . . 64
3.1.1 Chapter Contributions . . . 66
3.2 Protecting Information Assets with Morpheus . . . 66
3.2.1 Security Goals . . . 67
3.2.2 Threat Model . . . 68
3.2.3 Displacing Program Segments . . . 69
3.2.4 Encrypting Implementation Information Assets . . . 70
3.2.5 Re-Randomizing Morpheus' Protections with Churn . . . 70
3.3 The Morpheus Secure Architecture . . . 72
3.3.1 Hardware Architecture . . . 72
3.3.2 Churning Moving Target Defenses . . . 77
3.3.3 Support for Context Switching . . . 80
3.3.4 Compiling Protected Applications. . . . 81
3.4 Experimental Methodology . . . 81
3.5 Experimental Results and Evaluation . . . 82
3.5.1 Runtime Performance Overheads . . . 82
3.5.2 Area Overheads . . . 84
3.6 Security Analysis . . . 85
3.6.1 Penetration Testing . . . 85
3.6.2 Attacking Morpheus . . . 87
3.7 Discussion and Limitations . . . 89
3.8 Related Work . . . 90
3.9 Chapter Summary . . . 92

4 Strengthening Code Pointer Defenses via Fine-Grained Randomization . 93

4.1 Introduction . . . 94
4.1.1 Chapter Contributions . . . 96
4.2 Obfuscating Code Locations with Displaced and Dilated Address Spaces (DDAS) . . . 97
4.2.1 Security Goals . . . 98
4.2.2 Threat Model . . . 98
4.2.3 Displacing and Dilating the Code Segment . . . 99

4.2.4 Architecting an Efficient DDAS Implementation . . . 101
4.3 The DDAS Architecture . . . 103
4.3.1 Hardware Architecture . . . 103
4.3.2 DDAS Unit Microarchitecture . . . 104
4.3.3 Key Generation . . . 104
4.3.4 Context Switching Support . . . 105
4.4 Adding Runtime Re-Randomization (DDAS-R) . . . 106
4.4.1 Hardware Support . . . 107
4.4.2 Support for Context Switching in DDAS-R . . . 108
4.5 Experimental Methodology . . . 108
4.6 Experimental Results and Evaluation . . . 109
4.6.1 Runtime Performance Overheads . . . 110
4.6.2 Power and Area Overheads . . . 111
4.7 Security Analysis . . . 111
4.7.1 Qualitative Discussion of Security Against Common Attacks . . . 112
4.7.2 Quantitative Security Analysis of DDAS Entropy . . . 113
4.7.3 Attacking DDAS . . . 116
4.7.4 Performance-Security Trade-offs . . . 117
4.8 Discussion and Limitations . . . 118
4.9 Related Work . . . 119
4.10 Chapter Summary . . . 120

5 Redesigning Code and Pointer Protections to Use Encryption . . . 121

5.1 Introduction . . . 122
5.1.1 Chapter Contributions . . . 122
5.2 Redesigning Information Asset Protections in Morpheus II . . . 123
5.2.1 Security Goals . . . 124
5.2.2 Threat Model . . . 124
5.2.3 Encrypting Information Assets . . . 125
5.2.4 Isolating Encryption Keys . . . 126
5.2.5 Preventing Leakage with Randomization . . . 126
5.3 The Morpheus II Secure Architecture . . . 126
5.3.1 Hardware Architecture . . . 127
5.3.2 Compiling Protected Applications . . . 130
5.3.3 Differences From the Original Morpheus Design . . . 131
5.4 Experimental Methodology . . . 131
5.5 Experimental Results and Evaluation . . . 132
5.5.1 Runtime Performance Overheads . . . 132
5.5.2 Power and Area Overheads . . . 133
5.6 Security Analysis . . . 134
5.6.1 Qualitative Discussion of Security Against Common Attacks . . . 134
5.6.2 Bug Bounty Deployment . . . 135
5.6.3 CWE Evaluation . . . 135
5.6.4 Attacking Morpheus II . . . 136
5.7 Discussion and Limitations . . . 137

5.8 Related Work . . . 138
5.9 Chapter Summary . . . 140

6 Interlude: From Security to Privacy . . . 141

7 Architectural Approaches to Encrypting Data In Use . . . 143

7.1 Introduction . . . 144
7.1.1 Chapter Contributions . . . 146
7.2 Protecting Data In Use with Sequestered Encryption (SE) . . . 146
7.2.1 Security Goals . . . 148
7.2.2 Threat Model . . . 148
7.2.3 Novelty of Proposed Approach . . . 149
7.3 The Sequestered Encryption Architecture . . . 150
7.3.1 Hardware Architecture . . . 150
7.3.2 A Subtractive Approach to Enforcing our Security Goals . . . 154
7.3.3 SE Unit Microarchitecture . . . 154
7.3.4 Key Infrastructure . . . 155
7.3.5 Process-OS Interface . . . 156
7.4 Programming with Sequestered Encryption . . . 157
7.4.1 Abstractions for High-Level Languages . . . 157
7.4.2 Programming Limitations . . . 158
7.5 Experimental Methodology . . . 159
7.6 Experimental Results and Evaluation . . . 160
7.6.1 Overheads of Program Modifications . . . 160
7.6.2 Runtime Performance Overheads of SE . . . 162
7.7 Security Analysis . . . 163
7.7.1 Qualitative Discussion of Security of Evaluated Encryption Schemes 163
7.7.2 Resilience to Distinguishing Attacks . . . 163
7.8 Discussion and Limitations . . . 164
7.9 Related Work . . . 165
7.9.1 Homomorphic Encryption (HE) . . . 165
7.9.2 Privacy-Preserving ISA Extensions . . . 166
7.9.3 Trusted Execution Environments (TEEs) . . . 166
7.10 Chapter Summary . . . 167

8 Optimizing Encrypted Data Architectures . . . 168

8.1 Introduction . . . 169
8.1.1 Chapter Contributions . . . 169
8.2 Sequestered Encryption Optimized Microarchitecture . . . 170
8.2.1 Security Goals . . . 170
8.2.2 Optimized (Stateful) SE Unit Microarchitecture . . . 171
8.3 Experimental Methodology . . . 174
8.4 Experimental Results and Evaluation . . . 175
8.4.1 Runtime Performance Overheads of SE-Opt . . . 175
8.4.2 Comparison to Homomorphic Encryption . . . 177

8.5 Security Analysis . . . 178
8.5.1 Resilience to Timing Analysis . . . 178
8.5.2 Potential for Formal Verification . . . 179
8.6 Discussion and Limitations . . . 180
8.7 Chapter Summary . . . 180

9 Understanding the Program Implications of Encrypted Data Architectures 181

9.1 Introduction . . . 182
9.1.1 Chapter Contributions . . . 183
9.2 Data-Oblivious Transformations . . . 184
9.2.1 If-Conversion . . . 184
9.2.2 Loop-Conversion . . . 185
9.2.3 Access-Conversion . . . 186
9.2.4 Operators . . . 187
9.2.5 Frequency of Data-Oblivious Transformations . . . 187
9.3 Evaluation Methodology . . . 188
9.3.1 Compiler Support and Benchmarks . . . 188
9.3.2 Experimental Setup . . . 189
9.3.3 Metrics . . . 190
9.4 Characterization of Mechanized Data-Oblivious Transformations . . . 191
9.4.1 Performance Overhead of If-Conversion . . . 192
9.4.2 Performance Overhead of Loop-Conversion . . . 194
9.4.3 Performance Overheads of Access-Conversion . . . 195
9.4.4 Trends in Instruction Characteristics . . . 196
9.5 Algorithms for Data-Oblivious Execution . . . 197
9.5.1 String Search . . . 198
9.5.2 Integer Sort . . . 200
9.5.3 Element Distinctness . . . 201
9.5.4 Flooding Cells with Flood-Fill . . . 202
9.6 Discussion and Limitations . . . 203
9.7 Related Work . . . 204
9.8 Chapter Summary . . . 205

10 Conclusion . . . 206

10.1 Dissertation Summary . . . 208
10.2 Future Directions . . . 209

BIBLIOGRAPHY . . . 212

LIST OF FIGURES

FIGURE

1.1 **Patched WhatsApp Source Code for CVE-2019-3568.** Shortly after CVE-2019-3568 was disclosed, WhatsApp released a software update patching various buffer overflow vulnerabilities in the VoIP stack, including the above excerpt adapted from [3]. This patch added the check `size <= 0x5C8` to the else-if condition on line 6 to guard the vulnerable buffer copy. 3

2.1 **Program Life Cycle.** A programmer's intended finite state machine (FSM) is transformed into an implementation FSM through multiple phases of the program life cycle: high-level programming, compilation, and hardware realization. The first two phases produce intermediate FSMs, termed the language-level and instruction-level FSMs, respectively. Each phase employs an abstract model of computation to produce the resultant FSM. Additionally, each phase may contain other sources that add unintended state (grey nodes) and transitions (red edges) that are absent from the intended FSM. Undefined semantics in high-level programming and compilation introduce nondeterministic transitions (dashed red edges) that are defined later in the program life cycle. Within this framework, every input into Sigma (Σ) is a vulnerability source and introduces unintended states and transitions that can be leveraged for security exploits. 17

2.2 **Defense Techniques.** Vulnerabilities are exploited to transition the victim program from an intended state (white) to an unintended state (grey). Once outside of the constraints of the intended IFSM, an attacker can use knowledge of information assets to subvert security. Avoidance-based protections (2.2b) find and fix vulnerabilities to remove single transitions to unintended states. Enforcement-based protections (2.2c) take a complete approach by addressing *all* transitions of a particular type (*e.g.*, buffer overflows). Finally, obfuscation techniques (2.2d) conceal information assets and unintended states, preventing attackers from effectively and reliably learning enough information to synthesize an exploit. 36

3.1 **Example Control-Flow Attack.** This code exploits a buffer overflow vulnerability in `vulnerable()` to overwrite a return address. We note that the attack code utilizes undefined semantics associated with array overflows, stack frame organization, and code addresses. Morpheus randomizes values associated with undefined semantics to thwart malicious attacks while remaining benign to normal programs. 65

3.2 **Morpheus' Moving Target Defenses (MTDs).** At its core, Morpheus protects multiple *assets* from attackers, which are identified at runtime using *tagged memory*. The protections are layered on top of these assets: *displacement* and *encryption*. Finally, *churn* drives these two defenses to re-randomize them live. 67

3.3 **Thwarting Implementation Information Acquisition with Churn.** Exploits progress with the following phases: *probe*, *weaponize*, and *attack*. Ensembles of moving target defenses (EMTDs) increase uncertainty in key values needed to weaponize attacks, thus, probe times increase significantly. To ensure the patient attacker does not complete their probes, churn re-randomizes key program values at regular intervals. An attack detector senses when probe-like activity is occurring and reduces the churn period to strengthen defenses. 71

3.4 **Morpheus Architecture.** The Morpheus RISC-V extensions decrypt code and pointers at the L1-cache boundary ① and correct displaced pointers in the pipeline ②/③. The Churn Unit ④ supports re-randomization during runtime. Churn support also includes the Attack Detector and Threshold Register, which are now shown in this figure. The memory system is extended with 2-bit tags to track the domains of data values at runtime, while the Tag Propagator determines the tags for computed values. 73

3.5 **Address Space Displacement.** Address space displacement creates two displaced address spaces, DAS_D and DAS_C, shifted by a random displacement of d_{DP} and d_{CP}, respectively. Pointers are expressed in these superimposed address spaces but translated back to the VAS before use. With two displaced address spaces, any displacement can be used without creating collisions between code and data objects, increasing the entropy for the displacement keys. 76

3.6 **Domain Encryption.** Morpheus' encryption makes use of per-domain keys, denoted as k_C, k_{CP}, and k_{DP} for *code, code pointer,* and *data pointer*, respectively. These keys are selected by a value's domain tag and used to protect it in memory. Non-pointer data is unencrypted. 76

3.7 **Churning During Execution.** The churn cycle starts with ① a pipeline flush and ② new key generation and register updates. Next, ③ a threshold register is used to coordinate updating values in memory under the new displacements and encryption keys. When ④ churn completes, all domains have been updated, effectively disposing of any information the attacker may have acquired previously. 78

3.8 **Address Displacement Gate.** ① Regardless of if the system is churning or not, we always remove the most up-to-date d from the effective address before accessing the TLBs, since the pipeline will only generate requests with the newest d. ② When churning, the gate must determine if the value being loaded/stored is within a region yet updated by the Churn Unit or not. ③ If the pointer is being *loaded*, then it must be made "Clean" by adding Δd. On *stores*, the pointer may be made "Stale" by removing Δd. ④ Finally, if the pointer comes from the "Clean" region, the gate selects the value as-is without modification. 80

3.9 **Performance Overhead.** This graph shows the average and worst-case performance overheads of the chosen MiBench and C-based SPEC CPU2006 benchmarks, with churn periods varying from no churn (left) to continuous churn (right). The expected number of churn periods that would complete in the time estimated to penetrate Morpheus defenses, based on the attack study in Section 3.6, is shown below each churn period. Given these results, we target 50 ms as a "normal" churn period. 83

3.10 **SPEC CPU2006 Performance Overheads.** The graph shows SPEC CPU2006 performance overheads for varied churn periods. The continuous churn period is listed above the bar and differs based on the time needed to update all memory values used by the workload. 84
3.11 **Estimated Probe Times.** This figure shows the average estimated attack time to penetrate a de-featured Morpheus, which has fewer defense keys and no churn. Probe times are shown for Morpheus with (E) and without ($\overline{E}$) domain encryption, and with (P) and without ($\overline{P}$) pointer displacement. As more defenses are engaged, probe times grow, with fully-activated defenses requiring a probe time 5020x longer than the normal churn period (50 ms). 88

4.1 **DDAS Overview.** At load time, a 56 B function is displaced by 8 kB and dilated to 14.3 kB within the Displaced and Dilated Address Space (DDAS). Invalid memory regions are inserted between instructions and trigger security exceptions when accessed. Every 50 ms, the function layout re-randomizes, changing in both size and location. In DDAS, this function can be maximally inflated to 14 MB, and the entire address space can be dilated by over 250,000 times. DDAS addresses are derandomized before entering the virtual memory system, thereby eliminating performance impacts on the memory system. 95
4.2 **Basic DDAS Design.** The VAS is divided into segments of size S_{vas}. Each segment is mapped into DDAS and dilated by an i byte hole. The address space is shifted by d bytes. 100
4.3 **Table-Based DDAS Design.** Varying sized holes (i_0-i_n) are inserted between the valid addresses in a single VAS segment. These holes sum to i bytes. Additionally, each segment is divided into smaller ranges of size r to efficiently track the locations of holes in the Range-Map table. 100
4.4 **Functional Unit for Basic DDAS.** This unit computes Equations 4.2 and 4.3 in parallel for a given code pointer, A_{ddas}. 105
4.5 **Functional Unit for Table-Based DDAS.** This hardware unit computes Equations 4.5-c and 4.7-c. The Range-Map table, indexed by the range number, returns T for the A_{vas} calculation and T_r for the valid pointer assertion. 105
4.6 **DDAS(-R) Architecture.** The baseline RISC-V architecture is extended with a DDAS→VAS functional unit ① to translate code pointers during `JALR` instructions. The microarchitecture of these functional units are shown in Figures 4.4 and 4.5 (above). In DDAS-R, these functional units are extended to accommodate the mixed state of pointers during re-randomization. The DDAS Remapper ② uses tagged memory to identify and update stale code pointers. 107
4.7 **Average Performance Overheads.** The average overhead for the SPEC CPU2006 benchmark suite at varied re-randomization rates for the analyzed DDAS configurations. 110
4.8 **Performance Overheads for Table-Based, 2k DDAS.** The performance overheads per benchmark for a table-based, 2k-entry DDAS configuration, with 2 cycles of exposed latency. 111

4.9 **Entropy of Indirect Jumps.** The primary axis, in orange, shows the entropy of indirect jumps, where entropy is modeled as log_2 of the dilation in bytes. The secondary axis, in blue, shows the probability that a given key configuration will *not* impact the distance of an indirect jump. 114

5.1 **Morpheus II Architecture.** The Morpheus II secure architecture extensions serve to maintain the correctness of program execution when code pointers and code are encrypted. The Morpheus II extensions decrypt the target address for register-relative jumps ① and encrypt the return address for all control-flow redirections ②. Instructions are decrypted prior to the decode stage ③. These ciphers latencies fit within the current pipeline stages, minimizing runtime overheads. Compared with the previous Morpheus design (Figure 3.4), Morpheus II greatly reduces pipeline modifications and memory touchpoints. 128

7.1 **Sequestered Encryption Usage Model.** We consider a data owner who wants to take advantage of an online service but does not trust this service with their data. The data owner can encrypt their data and access key with a standard encryption technology in a way that only grants access to the trusted hardware within the data user's server. Regardless of whether the data user is honorable, the owner's data remains confidential. 145

7.2 **Simple Design of Sequestered Encryption Functional Unit.** A simple implementation of SE is an arithmetic logic unit (ALU) fitted with cryptographic engines. Data decryption, computation, and re-encryption are done as one atomic operation between two latches. The computation is selected by a public control signal. 147

7.3 **Sequestered Encryption Architecture.** Sequestered Encryption (SE) is architected as a functional unit that processes instructions belonging to the SE ISA Extension (Table 7.1). Thus, the x86-64 core is extended with an SE functional unit that computes on encrypted source operands ①. In addition, the register file is extended to accommodate 128-bit encrypted data types. No other modifications to the memory system are made. 151

7.4 **Encrypted Data Layout.** Primitive types (*e.g.*, `double`) are appended with an 8B random cryptographic salt and then encrypted to form a 128-bit encrypted data type (*e.g.*, `enc_double`). 153

7.5 **Sorting a List with Sequestered Encryption.** This program sorts a list of encrypted integers using SE operations, facilitated by our GNU C++ library. Due to the restricted SE environment (§7.4.2), a conditional move is used in place of an if-statement. 158

7.6 **Increase in Dynamic Instruction Count.** This figure shows the increase in dynamic instruction count due to data-oblivious program modifications (DO) and usage of the SE GNU C++ library, including additional loads for 128-bit data types (SE-ISA). . . 161

7.7 **Proportion of Dynamic Instruction Count.** This figure shows the proportion of dynamic instructions dispatched to the SE unit (*i.e.*, belonging to the SE ISA extension). On average, 30% of committed instructions were from the SE ISA extension. 161

7.8 **Performance Slowdowns for Sequestered Encryption.** This figure shows the overall performance slowdowns for the stateless SE microarchitecture (SE) when parameterized with 12-cycle QARMA, 20-cycle Simon, and 40-cycle AES encryption. Performance overheads across different ciphers scale almost linearly with the cipher's latency. 162

8.1 **Data Flow for Sequestered Encryption.** In the unoptimized microarchitecture, cryptographic latencies are exposed on every operation. Our optimizations bypass idempotent encrypt-decrypt operations and reduce latencies on the load path. 171

8.2 **Optimized Sequestered Encryption Microarchitecture.** The decryption cache and register cache forward plaintext values to dependent SE instructions, bypassing idempotent cipher operations on the critical path. Since these caches are indexed with non-secret information (*i.e.*, ciphertext values and register IDs), any timing variation due to these optimizations will not reveal any secret data. 173

8.3 **Performance Slowdowns for Sequestered Encryption with QARMA$_{11}$-128-σ_1.** This figure shows the overall performance slowdowns for the stateless SE microarchitecture (SE) and the optimized SE microarchitecture (SE-Opt) when parameterized with 12-cycle QARMA data encryption. The dashed lines represents slowdowns originating from software modifications alone. 176

9.1 **Implementation of CMOV.** To implement conditional moves using the x86 CMOVcc primitive, we emit inline `asm` statements that set the destination register `result` with either `v_true` or `v_false`, depending on the evaluation of the Boolean condition. . . 190

9.2 **Performance Overhead, Relative to Native (Log Scale).** This figure shows the performance overhead for each oblivious workload, relative to a native baseline. . . . 192

9.3 **Breakdown of Oblivious Runtime.** This figure shows the percentage of an oblivious workload's runtime attributed to different transformations. For example, 67% of the oblivious *Image-Filter* runtime is attributed to computation in the native algorithm, whereas 33% is attributed to if-conversion. *Edit-Distance* and *Kadane* experience speedups resulting from if-conversion, as noted in Section 9.4.1. *Bubble-Sort* also experiences a speedup from loop-conversion, which removes an early-exit condition, but this speedup is masked by the overheads of other data-oblivious transformations. . . 193

9.4 **Characterizing Trends in Oblivious Workloads.** This heatmap shows the trends in runtime performance and instruction characteristics across our twenty workloads. While dynamic instruction count consistently increases, branch count is varied, as if-conversion removes branch instructions, whereas other conversions add them. Instruction characteristics show improvement across multiple workloads. 197

9.5 **Total Cycle Count for Optimized Workloads (Log Scale).** This figure shows the overall processor cycle count for the native, oblivious, and optimized workloads discussed in our case studies on a log scale. Solid bars represent the native (unmodified) workload, whereas dashed bars represent the oblivious workload variants. Our study demonstrates how algorithm redesign can significantly reduce data-oblivious programming overheads. For example, *String-Search*'s oblivious performance overhead is reduced from 1214× to under 2×, while *Distinctness* is reduced from 767× to under 10×. All relative performance overheads are listed in Table 9.4. 199

LIST OF TABLES

TABLE

2.1 **Enumeration of Vulnerabilities.** Vulnerabilities are transitions to unintended states in the FSM. We classify vulnerabilities by their phase and source and provide examples of CWEs (child classes) belonging to each vulnerability class. In the text, we refer to this enumeration of vulnerabilities using the notation `V-1`. This list is not exhaustive of all vulnerabilities, but rather represents those most prevalent in security exploits. *The CWE enumeration can be slow to categorize emergent attacks. As a result, exploits that leverage mispeculation, such as LVI [4] (CVE-2020-0551), are often listed as having no associated CWE (written as NVD-CWE-noinfo). 28

2.2 **Enumeration of Implementation Information Assets.** We classify information assets by their phase and source, whether it is introduced during high-level programming via the notional computer model, during compilation via compiler choices, or during hardware realization via microarchitectural choices. Furthermore, we specify how this implementation information is acquired by attackers via documentation (DOC), reverse engineering efforts (ENGR), or probing techniques (PROBE). In the text, we refer to this enumeration of information assets using the notation `I-1`. This list is not exhaustive of all implementation information assets, but rather represents the classes most prevalent in security exploits. *Location of Objects originates from OS Choices when ASLR [5] is in place. 31

2.3 **Novel Vulnerability Sources in Control-Flow Attacks.** We enumerate control-flow attacks that have leveraged a novel vulnerability or information asset (*i.e.,* one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition. For space driven brevity, we adopt the following abbreviations: memory access error (MAE) and undefined semantics (U.S.). 42

2.4 **Chronological Ordering of Control-Flow Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent control-flow attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: memory access error (MAE), code pointer (CP), and data pointer (DP). 43

2.5 **Novel Vulnerability Sources and Acquisition Methods in Derandomization Attacks.** We enumerate derandomization attacks that have leveraged a novel vulnerability source or method of information acquisition (denoted by *). We classify these assets by their source and method of acquisition. In this genealogy, code location originates from OS Choices because ASLR [5] is in place. 47

2.6 **Chronological Ordering of Derandomization Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: memory access error (MAE), code pointer (CP), and data pointer (DP). 47
2.7 **Novel Vulnerability Sources in Timing Side-Channel Attacks.** We enumerate timing side-channel attacks that leverage a novel vulnerability or information asset (*i.e.,* one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition. For space driven brevity, we adopt the following abbreviations: microarchitecture (μArch). 50
2.8 **Chronological Ordering of Timing Side-Channel Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent side-channel attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviation: sharing-dependent (SD). 51
2.9 **Novel Vulnerability Sources in Transient Execution Attacks.** We enumerate transient execution attacks that have leveraged a novel vulnerability or information assets (*i.e.,* one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition. For space driven brevity, we adopt the following abbreviations: microarchitecture (μArch). 55
2.10 **Chronological Ordering of Transient Execution Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent transient execution attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviation: sharing-dependent (SD). 55
2.11 **Chronological Ordering of Defenses Presented in this Dissertation.** We enumerate the information assets protected by the defenses presented in this dissertation. Each asset is either protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: code pointer (CP), and data pointer (DP). 61

3.1 **Tag Propagation Rules.** This table lists the rules used to determine output tags (RD.tag) in the pipeline. These rules work like an if-else tree: the rules at the top are evaluated *first*, and if it does not apply, it moves to the next row in the table. The tag X implies that it can have any value and tag *Immed.* indicates that the second operand is an instruction immediate. `RD` is the destination register or destination memory location, in case of `ST`. 74
3.2 **Attack Detector Logic.** This table shows rules implemented by the attack detector. ABORT rules monitor particularly grievous operations and trigger an exception that terminates the program. CHURN rules detect undefined semantics that may be indicative of an ongoing attack. In response to these violations, the attack detector initiates a churn cycle. Tags are defined as follows: C = Code, CP = Code Ptr, D = Data, DP = Data Ptr. 75

3.3 **Experimental Configuration for gem5.** The specifications of our baseline RISC-V in-order core in gem5 [6]. A 4KB tag cache is added to the EMTD-enabled microarchitecture. 82
3.4 **Morpheus Hardware Estimation.** Numbers for SiFive cores were obtained from their website, for the U54 [7] and the E51 [8]. Caches are estimated using CACTI 7 [9]. QARMA units have a size of 1879.9μm^2 in 7nm [10]. This is scaled by 7.8 (as given in [11]) to get 0.0147mm^2 per unit in 32nm technology. 85
3.5 **Penetration Testing Results.** The Morpheus secure architecture was shown in simulations to stop all of the attacks listed. Additionally, all attacks trigger our attack detector. For each attack, the table shows how Morpheus stops the attack, with Morpheus domain encryption (E) and pointer displacement (P) applied to code (C), code pointers (CP), and data pointers (DP). 86
3.6 **Comparison Against Prior Works.** The assets protected by each work is listed. A ✓* indicates protection for only return addresses (for CP), or frame pointers (for DP). Displacing the code segment is listed as protection for code pointers (CP), rather than for code (C). For Fixed Interval systems, the average overhead is quoted at 50 ms. †TASR's overhead is originally 2.1%, however, as noted in Shuffler [12], TASR uses -Og, which can slow down SPEC benchmarks by ∼30% compared to -O2. 90

4.1 **Summary of DDAS Translation Equations.** This table summarizes the equations needed to implement basic or table-based DDAS. As detailed in Section 4.2.4, multiply, divide, and modulo operations are simplified by constraining the key parameters. The optimized/constrained equations are indicated via the notation "-c". 102
4.2 **Experimental Configuration for gem5.** The specifications of our baseline RISC-V out-of-order core in gem5 [6]. 109
4.3 **Hardware Estimation.** The area and power estimates for the analyzed DDAS configurations. 112
4.4 **Evaluation of DDAS.** The average inflation per instruction or page is the mean dilation incurred by a jump of that length in VAS. The table also includes the average percentage of memory occupied by untouchable holes and the calculated relative entropy (Section 4.7.2.2) per configuration. 113

5.1 **Morpheus II RISC-V ISA Extension.** Morpheus II adds three classes of instructions to support pointers: *i)* Encrypted pointer ALU operations, *ii)* Decrypted pointer relational tests, and *iii)* Decrypted unconditional jumps. 127
5.2 **Morpheus II FPGA Build Specification.** 132
5.3 **Performance Slowdown.** This table lists the recorded slowdown benchmarks running atop Morpheus II compared to an unprotected baseline. Differences between the two architectures are negligible, with a majority of applications actually experiencing a slight speedup (<0.01%) on Morpheus. This speedup can be attributed to system noise or the fact that Morpheus' re-randomization prefetches code and pointers into the cache. 133

5.4 **Runtime of Evaluated Database Queries.** We evaluated the performance overheads of various queries on our SQL database application deployed in the DARPA FETT challenge. The differences in execution time are within the bounds of run-to-run variations, demonstrating that Morpheus II has negligable performance impacts on the system. 134
5.5 **CWEs Evaluated on Morpheus.** This table lists CWE classes and information assets they exploit, denoted as representation of code (C), code pointers (CP), data pointers (DP), or data (D). Morpheus II did not stop CWEs that used data or data pointers, as it does not protect these assets. 136

7.1 **Sequestered Encryption ISA Extension.** The x86-64 ISA extension facilitates encrypted computation of arithmetic, relational, and Boolean operators and conditional moves within the SE unit. Loads and stores of encrypted values (`enc_mov`) are decomposed into two 64-bit `mov` micro-ops, as discussed in Section 7.3.1.2. 152
7.2 **Experimental Configuration for gem5.** The specifications of our baseline x86-64 out-of-order core in gem5 [6]. 159

8.1 **Experimental Configuration for gem5.** The specifications of our baseline x86-64 out-of-order core in gem5 [6]. 175
8.2 **Performance Overheads.** In this table, we show the slowdowns for our SE and SE-Opt microarchitecture designs when parameterized with AES-128 (40 cycles) [10], Simon-128/128 (20 cycles) [13], and QARMA$_{11}$-128-σ_1 (12 cycles) [10] data encryption. The second column reports slowdowns resulting from only the software modifications required to support SE (*i.e.*, SE encryption/decryption is disabled in the SE unit). . 177
8.3 **Comparison to Homomorphic Encryption.** This table presents the processor cycle count and relative performance overheads for the Porcupine kernels [14] when running on SEAL BFV and on SE-Opt parameterized with AES encryption. 178

9.1 **Oblivious Operators in C++.** This table lists the C++ operators included in our oblivious workloads. Short-circuited logical (`&&`, `||`) and ternary (`?:`) operators are compiled via control structures and therefore are not data-oblivious. 187
9.2 **Workloads.** Above, we list our evaluated workloads from VIP-Bench [15]. The ● symbol indicates which data-oblivious transformations are applied to each workload. We report average time complexity for the native and oblivious algorithm variants. The workloads are grouped depending on whether the data-oblivious transformations impact the asymptotic complexity. *The complexity of native *Primality-Test* depends on the value of n itself. 189
9.3 **Tuned Skylake Processor Configuration for gem5.** 190
9.4 **Case Studies.** Above, we list the benchmark variants for our four case studies: string search, integer sort, element distinctness, and flood fill. Overall, our algorithmic improvements provide a ∼86× geomean speedup compared to the mechanically converted oblivious workload. 198

ABSTRACT

There are increasing incidences of high-profile data breaches and clever new attacks that exploit weaknesses throughout the software stack, with recent attacks moving into the hardware layer (*e.g.*, Spectre [16] and Meltdown [17]). Yet, the security landscape consists not only of these novel exploits but also of exploits we have known about for decades that leverage vulnerabilities equally as pervasive. Despite the prevalence of these known vulnerabilities and significant efforts to defend against them, exploits remain widespread.

Understanding the landscape of security attacks can aid in the design of more durable defenses. Security attacks take on a similar structure, despite their diverse forms. Attackers leverage one or more vulnerabilities and system information assets to synthesize their exploit. Looking across attacks, we view these components as forming an inverted pyramid, with a small group of information assets leveraged alongside a larger group of vulnerabilities to commit an even larger number of exploits. Ultimately, classes of information assets that are instrumental for attacks (*e.g.*, pointers, data layout, cache organization) are lesser in number than that of vulnerabilities. Thus, by applying protections to a few critical pieces of information, defenses can potentially achieve broad coverage against security exploits.

Hardware provides an advantageous place to situate information asset protections as they can be applied systematically, regardless of program-level semantics. Namely, low-level hardware implementations isolate critical information assets from higher-level layers of the stack, providing broad coverage against exploits. Architectural approaches enable us to design *vulnerability-agnostic* systems, as any software running atop the architecture, including programs that contain vulnerabilities, are insulated from attack. Further, hardware-based approaches can often be optimized to enable more efficient implementations, reducing runtime overheads that degrade system performance.

Based on these insights, this dissertation explores how encrypted architectures—processors that encrypt information domains (*e.g.*, memory addresses, instructions, or data) directly in hardware—can provide comprehensive security and privacy guarantees. Our research has evolved from using encryption minimally in an ensemble of moving target defenses to comprehensively applying encryption with small but powerful architectural extensions. The first half of this dissertation studies the protection of code and pointers to

thwart control-flow attacks, following the evolution of the Morpheus secure architecture. The second half of this dissertation discusses how encrypted architectures can comprehensively protect sensitive data and be safely optimized. Vulnerability-tolerant design undercuts all our proposed defenses, as we aim to protect systems in the presence of pervasive software vulnerabilities. Further, we work toward employing strong encryption and building side-channel resilience while maintaining reasonable performance overheads. Our work demonstrates that architectural approaches can emerge as dynamic, expressive, and performant security and privacy solutions.

CHAPTER 1

Introduction

Your thesis title should be "Are we crazy enough to put encryption in the pipeline?" The answer is yes!
—Todd Austin

There are increasing incidences of high-profile data breaches and clever new attacks that exploit weaknesses throughout the software stack, with recent attacks moving into the hardware layer (*e.g.*, Spectre [16]). Yet, the security landscape consists not only of these novel exploits but also of exploits that we have known about for decades which leverage vulnerabilities equally as pervasive. Despite the prevalence of these known vulnerabilities and significant efforts to defend against them, exploits remain widespread. We seek to better understand the security landscape to disrupt this trend and design more durable solutions.

Security attacks take on a common structure, despite their diverse forms. Attackers leverage one or more vulnerabilities and system information assets to synthesize their exploit. Looking across attacks, we view these components as forming an inverted pyramid, with a small group of information assets leveraged alongside a larger group of vulnerabilities to commit an even larger number of exploits. Ultimately, classes of information assets that are instrumental for attacks (*e.g.*, pointers, data layout, cache organization) are lesser in number than that of vulnerabilities. Thus, by applying protections to a few critical pieces of information, defenses can potentially achieve wide coverage against security exploits.

In this dissertation, we explore how architectures that encrypt information domains (*i.e.*, code, pointers, and data) can provide comprehensive security and privacy protections with low performance overheads. This document details how our research evolved from using encryption minimally and layering multiple protection mechanisms in complex hardware designs to comprehensively applying encryption with small but powerful architectural extensions. The following section discusses the motivation behind this approach and summarizes the progression of the research presented in this thesis.

1.1 How Software Exploits are Constructed

The large and dominant class of ***software-driven security exploits*** are characterized by a specific sequence of software inputs that, when provided to the victim application, without any assistance from hardware modifications (*e.g.*, Trojans), triggers a vulnerability to cause some malicious and unintended behavior. These exploits are incredibly varied and often have diverse goals, such as installing malware or stealing secret data. Yet, we see a common structure. Software-driven security exploits are *constructed* using both vulnerabilities *and* information assets. Software vulnerabilities provide attackers an avenue to deliver an exploit, whereas information about the victim's system enables attackers to craft a malicious payload that produces their intended outcome.

The structure of security exploits is exemplified by a recent control-flow attack on WhatsApp's Voice over Internet Protocol (VoIP). Control-flow attacks are a class of security exploits in which remote attackers hijack program control to execute malicious code on a victim machine. Recently, control-flow attacks on the popular messaging platform WhatsApp have been linked to the installation of spyware on the phones of journalists and human-rights activists (CVE-2019-3568[1]) [18, 19, 20]. This exploit and its recorded uses are alarming; however, the vulnerability enabling this attack is rather mundane. Specifically, when processing call data in the VoIP stack, the WhatsApp application copied data into a fixed-size buffer without checking if that data fit *within* the buffer. Thus, if the data size exceeded the buffer, neighboring memory would be erroneously overwritten. This programming mistake is a ***vulnerability*** and belongs to one of the most common weakness categories: *buffer overflows* (CWE-122[2]) [21]. In WhatsApp, this vulnerability enables attackers to deliberately corrupt program memory at their whim.

While vulnerabilities provide an avenue for synthesizing a security exploit, they are not the only element required for an attack. To exploit the WhatsApp buffer overflow vulnerability, an attacker must construct a VoIP packet (phone call) that triggers the vulnerability and overwrites *specific* values in memory to gain control over the victim machine. Such construction is non-trivial and requires knowledge of the memory layout and the format of data, code, and pointers on the target device. Without this relevant system information, which we term ***information assets***, an attacker cannot accurately use the vulnerability

[1]Publicly disclosed cybersecurity flaws are assigned a unique Common Vulnerabilities and Exposures (CVE) identifier, which pertains to the particular instance of a vulnerability class leveraged for an attack. This list is maintained by the U.S. Department of Homeland Security and the Cybersecurity and Infrastructure Security Agency.

[2]Software and hardware vulnerabilities are classified by the community via the Common Weakness Enumeration (CWE). Each vulnerability class is assigned a unique CWE identifier. A CVE listing cites the CWE classes it belongs to.

```
1  void handle_incoming_traffic(_QWORD *inout_res, int IN_arg2, int size,
2                                                  int IN_arg4, int IN_arg5)
3  {
4      ...
5      // Conditional with added size check
6      else if(size <= 0x5C8 && ((*(_DWORD*)IN_arg2 & 0xFE00) == 51200 ) {
7        // Vulnerable buffer copy
8        inout_res = memcpy((char*) inout_res + 0x209F4,
9                           (const void*) IN_arg2,
10                          size);
11     }
12     ...
13 }
```

Figure 1.1: **Patched WhatsApp Source Code for CVE-2019-3568.** Shortly after CVE-2019-3568 was disclosed, WhatsApp released a software update patching various buffer overflow vulnerabilities in the VoIP stack, including the above excerpt adapted from [3]. This patch added the check `size <= 0x5C8` to the else-if condition on line 6 to guard the vulnerable buffer copy.

to perform remote code execution. ***Both vulnerabilities and information assets are necessary to construct a security exploit.*** Further, unlike vulnerabilities, which are assigned unique identifiers by the U.S. Department of Homeland Security (DHS) and the Cybersecurity and Infrastructure Security Agency (CISA), information assets are understudied and lack a formal categorization system. In Chapter 2, we develop a framework that captures how vulnerabilities and information assets exploited by attackers arise throughout the program life cycle. Our observations about the role of information assets have guided our approach to designing the security and privacy protections presented in this dissertation.

Stopping the WhatsApp Exploit. Defense strategies can be broadly categorized as *i)* Eliminating an instance of a vulnerability (CVE) used by that exploit, *ii)* Eliminating a class of vulnerabilities (CWE), or *iii)* Preventing the accumulation of information assets. In response to an attack, companies often immediately move to eliminating, or "patching," the CVE used in that exploit. WhatsApp CVE-2019-3568 was patched by adding a size check before the vulnerable buffer copy, as shown in Figure 1.1. While this approach stops the immediate threat, it is not a lasting way to protect against future attacks, as other vulnerabilities undoubtedly exist. Memory access errors, like the one used in CVE-2019-3568, are pervasive across code libraries and have accounted for 70% of vulnerabilities in Windows from 2006 to 2018 [21]. Indeed, shortly after CVE-2019-3568 was patched, another control-flow attack on WhatsApp was disclosed (CVE-2019-11932) that instead exploited a vulnerability in the code that parses GIF files.

1.2 The Role of Vulnerabilities and Information Assets

A small number of information assets have been, over time, used alongside a large number of vulnerabilities to commit an even larger number of attacks. Buffer overflow vulnerabilities (*e.g.*, CWE-122, as in the WhatsApp VoIP attack) have been acknowledged for more than two decades yet have been repeatedly leveraged alongside knowledge of a victim's memory layout to commit thousands of individual exploits. Further, adversaries have leveraged the same combination of information assets alongside different vulnerabilities (*e.g.*, CWE-415, as in the WhatsApp GIF attack) to commit even more attacks.

Vulnerabilities and information assets are challenging to address as they persist from the gap between the programmer's and the attacker's model of what happens during program execution.[3] A programmer's model of a system is limited to what the programming language offers. This abstraction enables programmers to write code without knowledge of the compiler or hardware. However, this setting also gives the attacker an advantage since their model can be much deeper, extending beyond the language to include the compiler, instruction set architecture, and physical processor implementation. It is *this* deeper understanding that enables them to exploit the programmer's software. For example, the C/C++ language leaves some behaviors undefined (*e.g.*, out-of-bounds accesses) and abstracts away runtime intricacies (*e.g.*, calling conventions). Programmers then may create code containing buffer overflows or other vulnerabilities due to ignorance of these undefined language semantics. This gap between the programmers understanding and the language provides attackers an avenue. With knowledge about a specific system implementation, the attacker can understand the materialization of the unspecified behavior and use other information assets (*e.g.*, the call stack layout) to construct a control-flow attack.

Understanding vulnerabilities, information assets, and this model gap can guide the development of security protections. Elimination of this gap is not a solution. Rather, this gap is innate to how we have implemented and designed systems and is ultimately necessary for humans to program and interact with complex technologies.

[3]This concept is related to weird machines [22], which model security exploits as transition system frameworks. In Chapter 2, we discuss how implementation information that arises from different points in the program life cycle contributes to this transition system. It is not the transition system but these pieces of information that interests us in this work.

1.3 Toward Durable Defenses

Knowledge of how security exploits are constructed and the relative role of vulnerabilities and information assets can guide the design of more durable defenses. Despite often having diverse attack goals, adversaries consistently leverage one or more vulnerabilities and system information assets to construct their exploit. Ultimately, classes of information assets that are instrumental for attacks (*e.g.*, pointers, data layout, cache organization) are lesser in number than that of vulnerabilities. Our study in Chapter 2 further reveals that a small amount of implementation information is used widely across numerous security attacks. Given these observations, defenses that focus on enforcing or obfuscating implementation information assets have the potential to be both effective and durable as they disrupt broad swaths of the attack landscape. In this dissertation, we explore how custom architectures can encrypt information assets to provide security and privacy protections. Below, we discuss the themes that inform the design of these defenses.

Advantages of Encryption. Mitigations that obfuscate information assets have been repeatedly subverted via derandomization attacks (§2.5.2). For example, Address Space Layout Randomization (ASLR) [5] obfuscates memory locations at load-time by randomizing the absolute location of distinct program segments. Despite its widespread adoption, ASLR can be trivially defeated on 32-bit systems since adversaries can quickly brute-force guess the randomized offset. Subsequent attacks have leveraged advanced information acquisition techniques to break 64-bit implementations of ASLR [23, 24, 25, 26]. Such attacks motivate the use of strong cryptography to protect information assets, as these schemes are considered highly resistant to cryptanalysis.[4] With strong cryptography, protected assets can be *completely exposed* to the attacker yet remain secret, assuming the attacker cannot acquire the encryption key.

Consideration of Side Channels. Even when cryptography is leveraged, attackers can sometimes still learn encrypted values by exploiting side channels—weakness where information propagates to externally observable states in an unintended manner. Many side channels are inherent to modern microarchitectural structures (*e.g.*, caches, branch predictor, *etc.*) and thus pose a significant security challenge. Ultimately, software-based approaches must be carefully designed to avoid side channels as they cede to the vulnerable hardware they run atop. Hardware-based protections can be built to possess side-channel resilience directly

[4]Cryptanalysis involves analyzing systems to decipher or guess an encrypted message without having access to the cryptographic key.

(*e.g.*, as done in Chapter 7), but this may be incur high performance overheads. Alternatively, periodic re-keying of obfuscation defenses has been employed to thwart information leakage, as any assets learned through side channels are nullified after the encryption key is refreshed. In this dissertation, we utilize both of these approaches to guard against side-channel attacks.

Benefits of Leveraging Hardware. The compiler and architecture dictate the form of critical information assets, like memory layout and pointer representation. Thus, these assets cannot be obfuscated via software protections alone. These assets must be protected in hardware, as it underlies all layers of the stack. Hardware defenses can isolate critical information assets from upper layers of the stack, effectively obfuscating the value of these assets from software running atop the architecture. Low-level hardware defenses enable the systematic protection of information assets, regardless of program semantics. Further, with architectural approaches, we can potentially design ***vulnerability-agnostic*** systems, as any software running atop the architecture, including programs that contain vulnerabilities, can be insulated from attack since the architecture directly thwarts the acquisition of integral system assets. Recent work, particularly Cryptographic Capability Computing (C^3) [27], demonstrates interest in incorporating encryption protections in hardware.[5]

Achieving Efficient Defenses. Cryptographic information asset protections are often dismissed due to anticipated performance loss. Indeed, the primary difficulty of designing encrypted architectures is that cryptographic schemes are costly, usually adding long delays to the critical path that severely impact a program's runtime. Many prior works carefully navigate the performance-security trade-off, introducing designs that minimize runtime overheads. For example, PointGuard [28], published in 2003, encrypts code pointers to thwart control-flow attacks like CVE-2019-3568. PointGuard only uses XOR encryption, citing that its single-cycle delay can fit neatly within a memory load delay slot. While PointGuard achieves minimal slowdowns, its encryption scheme can be trivially broken by known-plaintext attacks and frequency analysis, leaving it vulnerable to attack. Navigating performance-security trade-offs is a great challenge of security research, as I would argue many expect comprehensive, unbreakable protections with minimal cost. Custom hardware has the potential to reduce performance impacts, especially when employing encryption schemes specifically optimized for hardware implementations. The defenses presented early in this dissertation prioritize performance, making similar concessions to PointGuard (*e.g.*,

[5]I am delighted to be acknowledged in this paper after working with this group during an internship.

Chapter 5 uses a reduce-round cipher to minimize delays). With these concessions, our approaches have minimal runtime slowdowns. In later work, we embrace strong encryption holistically, considering performance optimizations afterward. This shift occurs as homomorphic encryption (HE) [29] is garnering more attention from the security community. Given that homomorphic encryption incurs many orders-of-magnitude slowdowns, we perceive that its adoption may indicate a shift toward deferring performance concerns in favor of comprehensive encryption.

1.4 Achieving Security and Privacy via Encrypted Architectures

In this dissertation, we explore how architectures that encrypt information domains (*i.e.*, code, pointers, and data) can provide comprehensive security and privacy protections with low performance overheads. Our research evolved from using encryption minimally in an ensemble of moving target defenses to comprehensively applying encryption with small but powerful architectural extensions. Throughout, we characterize the runtime performance of our proposed defenses and use these costs to inform our design.

Ensembles of Moving Target Defenses (EMTDs) for Security. Given that security exploits require knowledge of implementation information assets, a system can prevent attacks by deliberately introducing change and obfuscating system state to the point where it inhibits the synthesis of an exploit. Randomizing information assets as a defensive measure is also known as a *moving target defense* [30]. By randomizing the implementation information assets of a program each time it runs, attacks built for a system with static assets no longer work, forcing adversaries to probe for the unknown values needed for an exploit. The ***Morpheus Secure Architecture*** [31] adopts this approach by randomizing code and pointer assets to thwart control-flow attacks.[6] In its ensemble of moving target defenses, Morpheus randomizes code and data locations and encrypts code and pointers in memory. To obstruct the recovery of these assets through probing or other derandomization techniques, we periodically re-key and re-randomize Morpheus' moving target defenses in a process called ***churn***, effectively erasing any system information that may have been learned over the previous period. These asset protection mechanisms are integrated into the processor architecture to efficiently and effectively thwart control-flow exploits.

[6]Our study in Chapter 2 demonstrates that control-flow attacks require code and pointer assets.

Morpheus' *ensemble* provides its strength and novelty; security is achieved by obfuscating multiple information domains across layered defenses with runtime churn. However, the layers themselves are flawed; encryption is applied minimally due to performance concerns, rendering limited protection for pointers in the pipeline. Our subsequent research looks at ways in which we can strengthen information asset protections (represented by Morpheus' layers) to provide more durable security. Initially, we did this by developing more advanced address space randomization in ***Displaced and Dilated Address Space (DDAS)*** [32].[7] But, we soon realized that encryption offered a more powerful and resilient mechanism for protecting information assets. Our research later evolved to integrate strong cryptography in the processor architecture to provide more comprehensive security from attacks.

Embracing Strong Encryption. Morpheus' complex ensemble of defenses requires significant architectural modifications yet provides limited protection for pointers in the pipeline. We redesign Morpheus to leverage encryption more comprehensively. In ***Morpheus II*** [33], we bring code and pointer encryption from memory into the pipeline. Since encryption is more resilient to attack, we discard the other layers of Morpheus, eliminating the invasive microarchitecture changes[8] without compromising security. Morpheus II's strength at protecting information assets was tested in a recent industrial-grade bug bounty exercise where, over three months, more than five-hundred attackers were unable to break into a vulnerable application running atop the Morpheus II architecture [34]. The success of Morpheus II demonstrates the applicability and promise of encrypted architectures and motivates the consideration of this approach more broadly.

When designing Morpheus II, we are still incredibly performance conscious. Despite embracing encryption, we tweak our cryptographic scheme to use less rounds with the goal of eliminating latencies on the critical path (*e.g.*, code decryption now occurs in a single cycle that can be parallelized during instruction fetch). Our resulting performance overheads are negligible, but our protected assets are potentially vulnerable to cryptanalysis. In later work, our approach continues to evolve to embrace *strong* encryption despite its more significant overheads. This change is motivated, in part, by a pivot to protecting data privacy, where homomorphic encryption (HE) [29] has garnered attention despite its significant slowdowns. Following HE's lead, we work to develop protections that strongly encrypt sensitive data by leveraging the performant architectural approaches pioneered by Morpheus.

[7] Displaced and Dilated Address Space (DDAS) [32] obfuscates code locations and the relative distance between code objects to thwart control-flow attacks that manipulate code pointers (*e.g.*, by adding a constant offset), which were possible in Morpheus.

[8] Estimated area overheads drop form nearly 10% to 1.3% with the design of Morpheus II.

Pivoting to Data Privacy. Morpheus II demonstrated its durability against control-flow exploits[9] and other attacks that leverage code and pointers when it outlasted its peers in the three-month bug bounty program. Its effectiveness motivated us to consider other applications of encrypted architectures. Rather than protecting the assets needed to construct an exploit, we turned our attention to protecting the asset that many attacks aim to exfiltrate—sensitive data.[10] This shift further embodies our embrace of vulnerability-agnostic design, as vulnerabilities can be present in a system yet will not compromise the security of sensitive data.

Informed by the Morpheus defenses, we explore how encrypted architectures can provide comprehensive data privacy. We present ***Sequestered Encryption (SE)*** [35], a hardware capability that encrypts sensitive data in all software-accessible architectural state while enabling computation on this data. Conceptually, Sequestered Encryption functions as a small hardware enclave that evaluates instructions on encrypted data in the execute stage of the pipeline. This approach places encryption latencies on the critical path, inflating runtime overheads. However, we introduce classical architecture optimizations that effectively reduce performance overheads without compromising security. With these modifications, we find that overheads attributed to encryption latencies are only 17.5%, on average. The final chapter of this dissertation explores how programmatic changes can address the remaining overheads, marking Sequestered Encryption as an expressive and performant privacy mechanism.

1.5 Dissertation Contributions and Organization

Through the contributions summarized below, this dissertation provides evidence that encrypted architectures can emerge as expressive and performant security and privacy solutions. Our contributions illustrate how our research evolved from using encryption minimally in an ensemble of moving target defenses to comprehensively applying encryption with small but powerful architectural extensions. The first half of this dissertation (Chapters 3-5) studies the protection of code and pointers to thwart control-flow attacks, following the evolution

[9]The Morpheus threat models focus on guarding against control-flow attacks, despite their ability to impede *any* attack that requires knowledge of code or pointers. Morpheus can be subverted by attacks outside of the threat model we carefully constructed. This issue speaks to one of the biggest challenges in security defense design. The discipline expects that defenses provide full coverage over attacks in the targeted threat model, motivating researchers to define them narrowly. In practice, these defenses are valuable but must be combined with other security protections that have distinct threat models in order to fully insulate a system from attack.

[10]Data is indeed sometimes needed for the synthesis of an attack. However, this was not our motivation to encrypt data in subsequent work.

of the Morpheus designs. The second half of this dissertation (Chapters 7-9) discusses how encrypted architectures can comprehensively protect sensitive data.

- We provide a framework for organizing vulnerability sources introduced through the programming, compilation, and hardware realization phases of the program life cycle. We organize existing security attacks and defenses into this framework and discuss how vulnerabilities and information assets have evolved over time.

- We discuss how this framework can inform the design of security defenses, including how defenses can be made more durable by targeting knowledge of the implementation, termed information assets, rather than specific vulnerabilities.

- Under this guidance, we explore how architectures can provide security and privacy protections by obfuscating the implementation information needed (or targeted by) attacks. Our first secure architecture, named Morpheus [31], layers code and pointer protections in an ensemble of moving target defenses to thwart control-flow attacks. Specifically, Morpheus encrypts code and pointers in memory and applies address space displacement to pointers in the pipeline.

- While Morpheus is effective and performant, it requires significant architectural modifications and has limited protection for pointers in the pipeline. Our subsequent chapters serve to address these challenges. First, we propose extensions to Morpheus' address space displacement, termed Displaced and Dilated Address Spaces (DDAS) [32], that better protect pointers in the pipeline. We evaluate the security and performance of the DDAS architecture and find that it can detect 99.996% of pointer manipulations with minimal performance impacts.

- We then revisit the original Morpheus design and present ways encryption can be leveraged more comprehensively to minimize microarchitectural impacts without compromising security. We present Morpheus II [33], a secure architecture that processes encrypted code and code pointers in the pipeline, better protecting them from exfiltration. The strength of Morpheus II was demonstrated in a three-month bug bounty program, where over five-hundred attackers were unable to exploit a vulnerable application running atop the secure architecture.

- With the success of Morpheus II in preventing control-flow attacks, we began to consider how its approach can be leveraged to thwart other exploits. We turn our attention to encrypting sensitive data both as a way to discourage attacks and thwart data breaches. We present Sequestered Encryption (SE) [35]—a hardware capability that

provides comprehensive data privacy by removing all sensitive plaintext values from software-accessible architectural state.

- Unlike our previous architecture designs, Sequestered Encryption uses strong encryption (*e.g.* QARMA), placing long delays on the critical path. We leverage classical computer architecture techniques like caching to optimize the performance of Sequestered Encryption and avoid these long delays. We find that, with these optimizations, encryption latencies only account for 17.5% of runtime overheads, on average, with the remaining slowdowns originating from programmatic changes required to support encrypted architectures.

- We conclude this dissertation by investigating the software overheads accounting for the majority of the remaining performance overheads. We present a systematic characterization of the runtime overheads of encrypted data architectures and provide examples of ways programs can be designed to avoid these slowdowns.

- Throughout this dissertation, we design four encrypted architectures, characterize their runtime performance, and develop software frameworks to support these technologies. Our designs arc from reserving encryption protections for memory to embracing them comprehensively throughout the processor. Our iterative process embodies engineering design and ultimately demonstrates that encrypted architectures can be powerful and efficient security defenses.

Chapter 2 provides an overview of the landscape of software-driven security attacks and motivates the protection of information assets (*e.g.*, code, code pointers, and data) as a durable defense mechanism. Chapter 3 presents Morpheus [31], a secure architecture that protects code and pointers with an ensemble of moving target defenses. Chapter 4 iterates on Morpheus' code pointer protections by developing Displaced and Dilated Address Spaces (DDAS) [32] to further randomize code locations. In Chapter 5, we redesign Morpheus to leverage encryption more comprehensively and minimize microarchitectural changes [33]. The strength of this design was demonstrated during a three-month bug bounty competition where over five-hundred attackers were unable to penetrate a vulnerable application running atop the encrypted architecture. This result motivated us to consider how encrypted architectures can be leveraged to protect other information domains, such as sensitive data. We discuss this transition further in Chapter 6, where we pause for a brief interlude. This chapter denotes the split between the first half of this dissertation, which discusses encryption architectures that thwart control-flow attacks, and the second half of this dissertation, which discusses encrypted architectures that provide data privacy. Although these halves

have distinct objectives, their fundamental approach to achieving security is interconnected. Chapter 7 presents Sequestered Encryption [35], an architecture that encrypts sensitive data to provide comprehensive privacy guarantees in a low-trust setting. Chapter 8 refines this approach to reduce performance overheads without compromising security. Then, in Chapter 9, we characterize the overheads associated with the program implications (*i.e.*, data-obliviousness) of encrypted data architectures and show how algorithmic changes can enable performant privacy solutions. Finally, Chapter 10 concludes this dissertation and discusses future research directions.

CHAPTER 2

Background: Vulnerability Sources, Security Attacks, and Durable Hardware Defenses

A significant challenge in developing security defenses is that the space of security vulnerabilities and exploits is large and not fully understood. This results in specific point defenses that aim to patch particular vulnerabilities. While these defenses are valuable, they are often blindsided by fresh attacks that exploit new vulnerabilities. In this chapter, we study the organization of security vulnerabilities as they arise throughout the program life cycle in order to suggest ways in which we can make future defenses more durable. We classify these vulnerability sources through programming, compilation, and hardware realization, and show how each source introduces unintended states and transitions into the implementation. Further, we show how security exploits gain control by moving the implementation to an unintended state using knowledge of these sources, and how defenses work to prevent these transitions. We present illustrative case studies of four important attack genealogies—showing how they fit into the presented framework and how the sophistication of the exploits and defenses have evolved over time, providing us insights for the future. Specifically, these insights demonstrate that protecting information assets, such as code and code pointers, can protect against large classes of security exploits. Throughout the remainder of this dissertation, we employ this approach and leverage trusted hardware to defend information assets efficiently and effectively.

The research presented in this chapter appears in:

[1] Lauren Biernacki, Mark Gallagher, Zhixing Xu, Misiker Tadesse Aga, Austin Harris, Shijia Wei, Mohit Tiwari, Baris Kasikci, Sharad Malik, Todd Austin. ***"Software-Driven Security Attacks: From Vulnerability Sources to Durable Hardware Defenses,"*** ACM Journal on Emerging Technologies in Computing Systems (JETC), Volume 17, Issue 3, 2021. DOI: 10.1145/3456299.

2.1 Introduction

There is an increasing body of work in the area of hardware defenses for software-driven security attacks.[1] A major challenge in developing these defenses is that the space of security vulnerabilities and exploits is large and not fully understood. This results in specific point defenses that aim to patch particular vulnerabilities. While these defenses are still valuable, they are often blindsided by fresh attacks that exploit new vulnerabilities. We believe that there is value to be gained by organizing the sources of these vulnerabilities and understanding how they lead to specific attacks and defenses.

At an abstract level, security exploits prevail due to a mismatch between the programmer's intended finite-state machine (FSM) for an application and its concrete implementation on a general-purpose processor. Through the process of programming, compilation, and hardware realization, there are various sources of security vulnerabilities. Each of these sources introduces states in the concrete implementation that were not intended by the programmer. To synthesize a security exploit, attackers first leverage a vulnerability to move the concrete implementation to a state external to the programmer's intent. Now, outside of the constraints of the original program, an attacker can manipulate the finite-state machine to subvert security guarantees. This process requires accurate knowledge of implementation information and the state space to be successful.

In this chapter, we classify the sources of vulnerabilities and implementation information assets leveraged by security attacks. Accordingly, we discuss how defenses directly work to address these sources of exploits—through thwarting access to the information assets and the implementation state space. To support our formalization, we present case studies of prominent attack genealogies, including control-flow attacks that leverage program-level vulnerabilities to affect execution. Furthermore, we show that side-channel attacks have relied heavily on microarchitectural implementation information and that these two techniques have been combined to break some of the field's most sophisticated defenses. These case studies also serve to provide background for the work presented in this dissertation. Our study reveals that, while attacks evolve quickly, the vulnerability sources and their impact on the final state space of the implementation evolve much more slowly. These observations suggest that defenses that target the connection between vulnerability sources and the implementation state space may be more effective and durable than traditional security approaches that focus on finding and fixing vulnerabilities.

[1] Software-driven exploits are characterized by a specific sequence of software inputs that, when provided to the victim application, without any assistance from hardware modifications (*e.g.*, Trojans), triggers a vulnerability to cause some malicious and unintended behavior.

2.1.1 Chapter Contributions

This chapter makes the following contributions:

- We provide a framework for organizing vulnerabilities that are introduced through the programming, compilation, and hardware realization phases of the program life cycle.
- We show how vulnerabilities result in mismatches between the programmer's intended FSM and the implementation FSM. In addition, we present how the notion of auxiliary states in the implementation FSM can capture side-channel information as well as analog effects due to charge leakage exploited in attacks like Rowhammer.
- We discuss how security exploits fit within this framework by transitioning the implementation FSM to an unintended state. Further, exploiting a vulnerability requires knowledge of information assets throughout the program life cycle and the state space of the resultant implementation FSM.
- We propose a classification for security defenses based on how they protect the implementation state space. We discuss how defenses can be made more durable by targeting the implementation state space rather than specific vulnerabilities.
- Finally, we present case studies of the following prominent attack genealogies and show how they fit into the presented framework. These case studies illustrate how the exploits and defenses have evolved over time, providing insights for the future:
  - ***Control-Flow Attacks***—Hijack the victim's control flow to execute malicious code via either control plane (*i.e.*, code pointer) or data plane (*i.e.*, data pointer) manipulations
  - ***Derandomization Attacks***—Leak implementation information of a randomization defense in an effort to subvert the protection
  - ***Timing Side-Channel Attacks***—Leak a program secret by measuring timing information
  - ***Transient Execution Attacks***—Leverage mispeculation to induce unintended execution and exfiltrate a program secret

This chapter is organized as follows. Section 2.2 presents the life cycle of a program and outlines the vulnerability sources at each phase in this life cycle. Section 2.3 discusses how vulnerability sources throughout the program life cycle are leveraged for attacks. Section 2.4 characterizes how defenses work within our framework to mitigate attacks. Section 2.5

presents case studies of four important attack genealogies—control-flow attacks, derandomization attacks, timing side-channel attacks, and transient execution attacks—and shows how they fit into our framework. Sections 2.6-2.8 conclude this chapter by highlighting trends and takeaways from this work and discussing how the insights provided by this chapter influence the work in this dissertation.

2.2 Modeling the Program Life Cycle

We can model the execution of a program as a transition system that morphs through the phases of programming, compilation, and hardware realization to realize the intent of the programmer (Figure 2.1). Before writing a line of code, a programmer has some notion of their intended software design. We model this desired program functionality using a finite-state machine (FSM) that captures the programmer's intent, termed the ***intended FSM***. This FSM exists only in the mind of the programmer and captures program behavior as transitions between abstract states. The programmer eventually *programs* hardware to emulate the functionality of the intended FSM. This act of programming requires multiple transformations to the intended FSM, ultimately yielding an ***implementation FSM*** that runs on the target processor. The gap between these two transition systems serves as the origin for security exploits.

Previous work has similarly presented the intended FSM to formalize disparities between the programmer's intent and the final implementation [22]. We take this a step further by understanding the transformations across multiple ***phases of the program life cycle***, shown in Figure 2.1: high-level programming, compilation, and hardware realization. The first two phases produce intermediate FSMs, termed the language-level and instruction-level FSMs, respectively. Each phase employs an abstract *model* of computation to produce the resultant FSM and may contain other sources that add state and transitions not in the intended FSM (referred to as unintended states and transitions). These unintended states added throughout the program life cycle are dangerous, as cleverly crafted inputs that reach them can force the program outside of the original specification. Once no longer bounded by the program's constraints, the implementation may produce unexpected results or violate security guarantees. Thus, these sources of unintended states are effectively ***vulnerability sources***, shown in blue in Figure 2.1. Below, we describe these phases and the vulnerability sources contained in each.

Our program life cycle framework addresses the significant challenge of understanding the space of security exploits by presenting a conceptual model for organizing vulnerability sources and modeling how defenses target these sources. By representing programs as

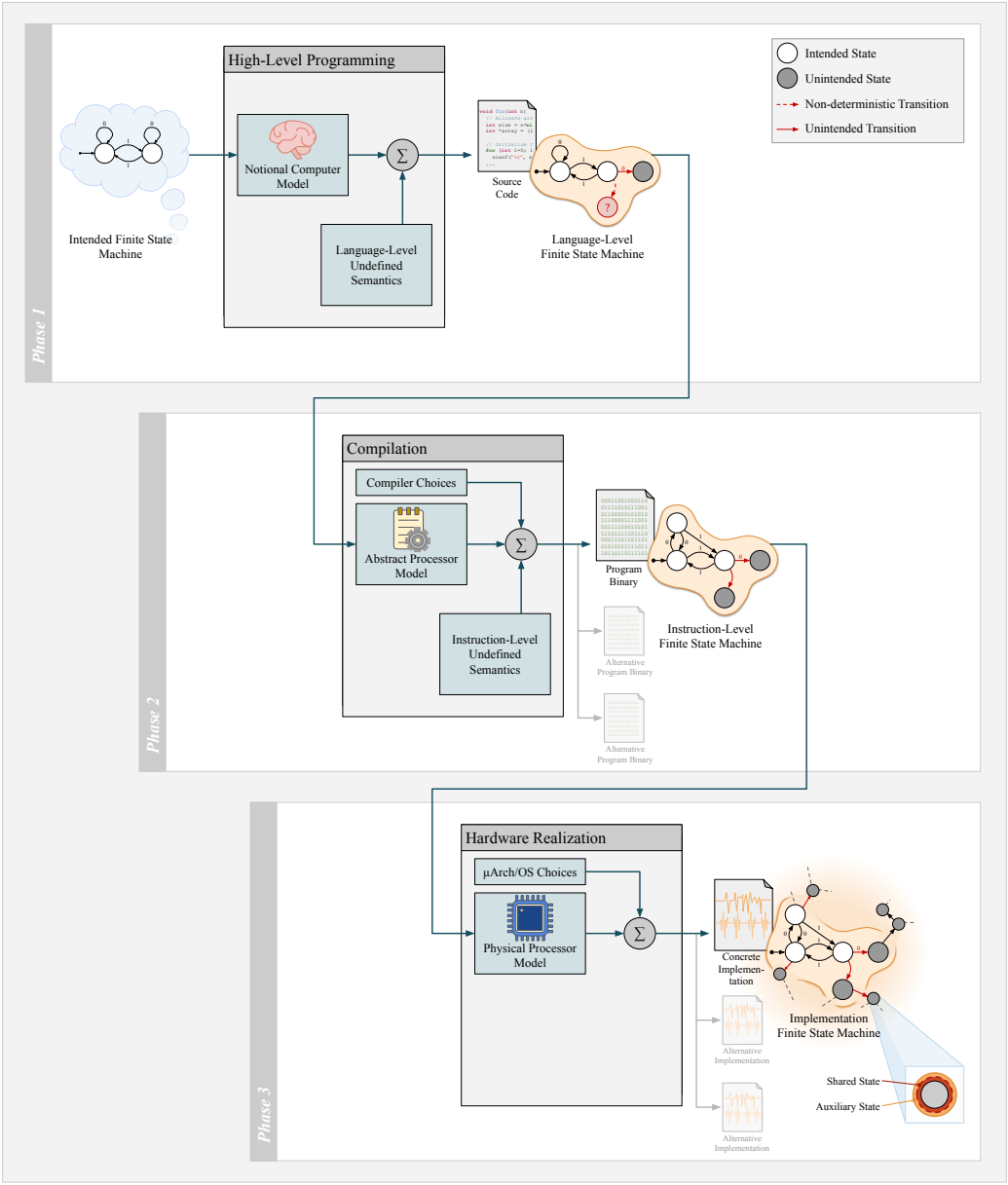


Figure 2.1: **Program Life Cycle.** A programmer's intended finite state machine (FSM) is transformed into an implementation FSM through multiple phases of the program life cycle: high-level programming, compilation, and hardware realization. The first two phases produce intermediate FSMs, termed the language-level and instruction-level FSMs, respectively. Each phase employs an abstract model of computation to produce the resultant FSM. Additionally, each phase may contain other sources that add unintended state (grey nodes) and transitions (red edges) that are absent from the intended FSM. Undefined semantics in high-level programming and compilation introduce non-deterministic transitions (dashed red edges) that are defined later in the program life cycle. Within this framework, every input into Sigma (Σ) is a vulnerability source and introduces unintended states and transitions that can be leveraged for security exploits.

transition systems, we can reason about security vulnerabilities as unintended transitions within the state space of program evolution (*i.e.,* the program's FSM), where state is characterized by what the program is doing (*i.e.,* data state, like memory contents) and where the program is in execution (*i.e.,* control state, like the program counter). Similarly, we can model defenses as preventing a program from making these unintended transitions or limiting subsequent unintended transitions that can be forced by an attacker to induce malicious functionality. At each stage in the program life cycle, it is useful to think of the state of the program and what causes it to move away from its intended next state to one where it is susceptible to an attack.

2.2.1 High-Level Programming

High-level programming is the process of expressing the programmer's intended application into source code in some programming language. High-level programming modifies the intended FSM, concretizing potentially abstract states, expanding the state space, and introducing new transitions. Two distinct mechanisms drive this modification: the programmer's notional computer model and language-level undefined semantics.

2.2.1.1 Notional Computer Model

A ***notional computer model*** is specific to a programmer and captures all assumptions they hold of the computation engine, including notions of the programming language, compiler, and processor. The specifics of this model depend heavily on the programmer's expertise, with a novice programmer's notional computer model likely capturing only a subset of the operations that the programming language affords. On the other hand, an expert programmer's notional computer model is likely to span beyond the language to capture elements of the entire system. For example, a programmer with knowledge of the target processor's memory layout may manually attempt structure packing to reduce their program's memory footprint. While a programmer may have different notional computer models specific to each programming language, this distinction is not relevant to the ideas presented in this work.

The notional computer model permits the introduction of logic bugs and vulnerabilities into the resultant source code. These flaws can arise from discrepancies between the programmer's notional computer model and actual program execution. For example, consider a novice programmer performing array operations in C who is unaware that C uses zero-based indexing. To access the first element of an array of length $n \geq 1$, they load index one, rather than index zero. While the programmer intends to access the first element of the array, they mistakenly load the second. This operation is unintended with reference to the

intended FSM, introducing a transition to an unintended state in the resultant ***language-level FSM***. In this example, the result of this transition is well defined within the high-level programming phase. However, other transitions introduced within this phase may be non-deterministic, only to be defined later in the program life-cycle.

2.2.1.2 Language-Level Undefined Semantics

Not all programming mistakes result in well-defined transitions in the language-level FSM. Non-deterministic transitions are also introduced during the high-level programming phase due to the effects of ***language-level undefined semantics***. In this work, we define language-level undefined semantics as any source code construct for which the programming language standard does not impose a specific behavior. Continuing the above example, to access the last element of the array, the programmer loads index n, a nonexistent item outside of the array's bounds. Out-of-bounds memory accesses are undefined in the C-language specification, meaning they are permitted but their effect is not guaranteed. This operation is unintended with reference to the intended FSM, and its result is *undefined*, introducing a non-deterministic transition (unspecified destination state) in the resultant machine. Only after compilation and hardware realization will we know the destination state of this transition.

Language-level undefined semantics comprise the traditional notions of undefined behavior, unspecified behavior, and implementation-defined behavior in the C/C++ standard. For undefined and unspecified behavior, the standard imposes no deterministic outcomes—for undefined behavior, the standard states nothing, whereas, for unspecified behavior, the standard states at least two different outcomes. Implementation-defined behavior is a subset of unspecified behavior that the operating environment implements consistently and documents fully (*e.g.*, whether `char` behaves as `signed char` or `unsigned char`). Programmers with knowledge of the target system can reliably leverage implementation-defined behavior due to the environment's guarantees, whereas the usage of undefined behavior is generally regarded as erroneous. We classify all of these three distinctions as language-level undefined semantics because the use of these semantics, either unintentionally or intentionally, introduces non-deterministic transitions into the language-level FSM.

2.2.1.3 Phase Summary

The high-level programming phase produces source code that provides a specification of a language-level FSM. This resultant FSM emulates the intended FSM and includes an expanded state space due to the effects of the notional computer model and language-level

undefined semantics. Specifically, the notional computer model introduces unintended states and transitions into the language-level FSM due to misconceptions held by the programmer (*e.g.,* confusion between zero- and one-based indexes). Language-level undefined semantics add non-deterministic transitions that are only defined during the later phases of compilation or hardware realization (*e.g.,* buffer overflow). Attackers leverage knowledge of these effects as they provide a direct means to invoke transitions to unintended states to gain control or leak information.

2.2.2 Compilation

Compilation transforms source code into a program binary for the target processor. Compilation modifies the language-level FSM, resolving its non-deterministic transitions, expanding the state space, and introducing new transitions, using three mechanisms: the compiler's abstract processor model, compiler choices, and instruction-level undefined semantics.

2.2.2.1 Abstract Processor Model

The ***abstract processor model*** captures the target processor and context in which the program executes to permit code generation and optimization. It includes notions of the instruction set architecture, such as the number of registers or addressing modes on the target device, needed to transform the source code into a form that can be executed on the target processor. This information is used by the abstract processor model to make decisions regarding register allocation and instruction scheduling. Additionally, the model also captures the functionality of the target device, which is necessary to achieve correctness while performing optimizations. By modeling the semantics of the source and target language, the compiler can assure that any transformations to the FSM maintain input and output consistency (*i.e.*, the program still produces the intended output for possible inputs). An abstract processor model is specific to a compiler instance and can produce alternative program binaries dependent on the decisions made during compilation.

The abstract processor model may be incomplete or even faulty, introducing unintended functionality into the resultant binary. For example, while compilers model program functionality, many do not model microarchitectural state. This disparity permits optimizations that maintain input-output consistency but modify cache contents or memory access patterns. One culprit of this memory alteration is dead store elimination. Consider an application that reads a password from the command line, computes its hash, and sends the hash to the server for authentication. During the calculation of the hash function, the plaintext password is stored in memory. Afterward, the program erases this variable for security purposes

by setting it to zero. From the compiler's perspective, the password is never used after it is set to zero, meaning that this zeroing store is dead. Thus, through dead store elimination, the compiler removes this zeroing store, causing the true plaintext value to persist in memory despite the programmer's intent. With respect to our framework, the compiler introduces unintended states and transitions into the resultant ***instruction-level FSM*** where the program counter advances and the plaintext password persists in memory. This program state of the incremented program counter and sensitive password is not intended by the programmer. Ultimately, discrepancies in the abstract processor model introduce unintended functionality to the resultant FSM and have been termed the "correctness-security gap" in compilers [36].

2.2.2.2 Compiler Choices

Compiler choices encompass the decisions made by the compiler to implement source code constructs on a hardware platform. Most notably, compiler choices include data and code layout and representation. These decisions reflect the abstract processor model, as the instruction set and target architecture limit viable options. Compiler choices affect the resultant program binary and instruction-level FSM by adding complexity, increasing the size and number of states, and modifying transitions. Compiler choices may also resolve non-determinism in the language-level FSM, for example, by specifying that the type `char` behaves as `unsigned char`.

We find that attackers repeatedly leverage knowledge of compiler choices to perpetrate security exploits. Consider the buffer overflow example discussed in Section 2.2.1. While this vulnerability is introduced during the high-level programming phase, it can only be exploited for a stack smashing attack with knowledge of compiler choices. Namely, an attacker must know the contents of the stack as well as the target function's location to overwrite the return address and redirect program execution reliably [37]. These two attack assets are determined by compiler choices.

2.2.2.3 Instruction-Level Undefined Semantics

Akin to high-level programming languages, assembly languages may include constructs with no specific defined behavior, termed ***instruction-level undefined semantics***. These undefined semantics within the assembly language similarly add non-deterministic transitions to the resulting instruction-level FSM. For example, the x86-64 standard does not define the value of flags for all instructions. The parity flag, `%pf`, is undefined for the `andn` instruction, enabling processors to either update the flag or keep the previous value. Previous work has shown that processors implement this semantic differently [38]. While instruction-level

undefined semantics cause unintended states and transitions in the implementation FSM, we have not found evidence of existing security exploits that leverage these assets.

2.2.2.4 Phase Summary

The compilation phase produces a program binary file that provides a bit-level specification of an instruction-level FSM. This binary is one of many possible alternative files that can be generated depending on the compiler instance and choices. The instruction-level FSM speaks to these choices, capturing all compiler decisions, including data layout. Hence, the resultant transition system includes an expanded state space due to the effects of compiler choices and the abstract processor model. The abstract processor model introduces both unintended states and transitions into the resultant instruction-level FSM (*e.g.,* persistent state security violations). Instruction-level undefined semantics add non-deterministic transitions that are later determinized by the microarchitecture. Attackers leverage knowledge of these effects, especially code and data layout, to perpetrate security exploits on the target hardware platform.

2.2.3 Hardware Realization

Hardware realization is a process that takes input in the form of a program binary and executes it on a hardware platform. This program execution follows the ***implementation FSM*** which captures all reachable states in the target processor, including modeling the relevant contents of microarchitectural structures. Hardware realization resolves all remaining non-determinism in the implementation FSM, expands its state space, and introduces new transitions, under the influence of two mechanisms: the physical processor model and microarchitectural choices.

2.2.3.1 Physical Processor Model

The ***physical processor model*** depicts one of many possible architecture targets for the resultant implementation, modeling the hardware that implements the functionality specified by the instruction set architecture. For example, the physical processor model transforms load/store operations in the program binary into complex hardware-based protocols that implement virtual memory. Because the physical processor model includes notions of all microarchitectural structures, it expands the state space of the instruction-level FSM immensely. The resultant implementation FSM contains states representing all possible combinations of the contents of these structures that are reachable from the initial state (*e.g.*, all possible values of the program counter).

The physical processor model may also perform transformations on the FSM to optimize performance, such as realizing speculative execution. These transformations preserve functional correctness but can modify state in an unintended manner, increasing the complexity of the resultant state space and introducing new transitions among states. The most compelling and currently relevant example of this phenomenon is mispeculation. During branch misprediction, modern processors squash in-flight instructions and roll back execution. These procedures preserve program functionality. However, not all changes to the microarchitecture are reverted. Evidence of the mispeculated instructions remains in structures like the cache, a property that recent attacks such as Spectre [16] and Meltdown [17] have exploited to exfiltrate program secrets. In terms of the implementation FSM, the state before and after rollback is *different* even though functional correctness is preserved.

In isolation, mispeculation is not dangerous. However, it becomes dangerous when combined with other effects of the physical processor model. Specifically, given the current design of microarchitectural structures, the physical processor model introduces state to the resultant FSM that is shared between programs. This ***shared state*** originates from structures that are not flushed during context switching, allowing data to linger after execution completes. In the context of the implementation FSM, this shared state manifests as residues from other programs that can affect the victim program's execution, or alternatively be used to monitor the victim program externally. Shared state can be dangerous as it establishes a channel for third-party observation of a program and may even permit external influence over execution. In the case of mispeculation, shared state enables an attacker to observe the residue of squashed instructions through the cache. Shared state also allows an attacker to influence which control transfers are mispredicted by priming the branch predictor. These mechanisms have been abused for devastating hardware-based exploits [16, 17].

2.2.3.2 Microarchitectural and OS Choices

Microarchitectural choices encompass decisions that realize microarchitectural structures on the target device, such as the size and layout of the caches, TLB, store buffer, etc. There may be multiple microarchitectural choices that implement the functionality required by the physical processor model, but ultimately only one design is employed, affecting the resulting state space in the implementation FSM. We distinguish microarchitectural choices from the physical processor model because attacks often leverage these specific details in security exploits. For example, shared cache state can be used to observe transient instructions, as mentioned above. However, an attacker must have knowledge about the cache structure, including the size and associativity of the cache, to extract this information with high fidelity. This knowledge of the cache originates from microarchitectural choices. While many attacks

are still viable without knowledge of microarchitectural choices (*e.g.,* control-flow attacks), we find that timing side channels and other architecture-based attacks repeatedly leverage knowledge of these decisions. ***Operating System (OS) choices*** capture decisions made by the system during runtime. This mechanism primarily represents code and data locations determined at load time due to the use of either position-independent code or Address Space Layout Randomization (ASLR) [5]. In this work, we assume code and data location is determined at compile time unless explicitly stated.

2.2.3.3 Auxiliary State

The implementation FSM also includes ***auxiliary state***, such as wall clock time and occurrence count, that is not reflected at the bit-level in the microarchitecture but retains information and may influence transitions during execution on the target processor. Notably, analog effects can be discretized as auxiliary state for the purpose of modeling them in the FSM. In this work, we distinguish between readable and writeable auxiliary state. Prominent examples of auxiliary states leveraged by security exploits include overall program execution time [39] and occurrence count of memory accesses, which can be used to capture DRAM cell charge leakage [40]. Program execution time is a readable auxiliary state that can be "read" in a variety of ways by an attack. Occurrence count is a writeable auxiliary state that reflects the count of a particular operation (*e.g.,* memory access) and which, on crossing some threshold in some duration, leads to a change of memory state (*e.g.*, bit flip in a DRAM). Thus, occurrence count can be used to effectively model the analog effects of the microarchitecture. As this information is not reflected in the microarchitecture directly, auxiliary state is, in general, hard to model in advance and instead modeled *post-facto* once its possible use is known.

2.2.3.4 Phase Summary

The hardware realization phase produces an implementation FSM that runs on a specific target architecture (of many) and sufficiently emulates the intended FSM. The implementation captures all additional state resulting from the physical processor model and microarchitectural choices, as well as the prior compilation and high-level programming phases. The physical processor model may also introduce transitions that do not match the programmer's intent, such as leaving execution residues in microarchitectural structures. Finally, the hardware-implemented FSM contains no non-deterministic transitions, as every transition can be fully defined given knowledge of the entire state space. The hardware realization stage produces an implementation that contains numerous unintended states and transitions

resulting from the effects of the entire program life cycle. Knowledge of these mechanisms is leveraged across attack classifications to perpetrate security exploits.

2.2.4 Relation to Previous Works

The framework presented in this chapter details how the different phases of the program life cycle introduce unintended states and transitions into the final implementation. This framework builds upon the *weird machines* model that captures how security exploits arise from unexpected program input that triggers unintended functionality [22, 41]. In that work, reachable states in the hardware implementation that enable unintended functionality form a new computational device, termed a "weird" machine, that permits security exploits. Work on weird machines specifically looks at the disparities between the programmer's intent and the resultant hardware implementation, *but not the intermediate mechanisms at play.* Dullien has characterized these differences using formal computational models [22]. Other work has studied specific instances of weird machines in the context of page faults [42], ELF executables [43], and proof-carrying code [44].

There is a significant body of work that identifies how undefined semantics in the C and C++ programming languages introduce vulnerabilities during compilation [36, 45, 46, 47, 48]. These works document how choices made during compilation resolve non-determinism from language-level undefined semantics in an unexpected way, such as by removing security checks (Section 2.3.1.2) [36]. Other work has modeled how weird machines arise during compilation due to behaviors in the target language that cannot be achieved by the semantics of the source language [49].

No work has provided a complete framework of how programmer conceptions, compiler interactions, and microarchitectural details introduce unintended states and influence security exploitation. In this work, we broadly reassess security exploits through the complete program life cycle, expanding the weird machine model to capture effects from the programmer, compiler, and microarchitecture. Our goal is threefold: *i)* To capture *all* sources of vulnerabilities throughout the program life cycle, *ii)* To understand how these sources are exploited to force the implementation FSM into an unintended state that compromises security, and *iii)* To illuminate how knowledge of this framework and the implementation FSM can lead to an understanding of durable defenses that thwart complete categories of unintended functionality rather than patching individual vulnerabilities.

2.3 Security Exploits

In Section 2.2, we presented our program life cycle framework, which models how high-level programming, compilation, and hardware realization introduce unintended states and transitions into the implementation FSM. Unintended states permit security exploits by enabling an attacker to circumvent the programmer's intent and cause unwanted program behavior. In this section, we delve further into how security exploits rely on these states to infiltrate victim programs.

Within our proposed framework, we characterize a ***security exploit*** as a path from intended to unintended states that produces unexpected and unwanted behavior in the implementation that is not possible in the intended FSM. Perpetrating an exploit requires two critical components: a transition between an intended and unintended state, termed a ***vulnerability***, to bypass the program intent/specification, and knowledge of the state space, termed ***implementation information assets***, to reliably manipulate the implementation FSM. We have found that security exploits leverage vulnerabilities and information assets from all phases of the program life cycle. These ***vulnerability sources*** are the blue colored boxes in Figure 2.1. We note that numerous exploits for a single vulnerability exist if multiple paths in the implementation FSM can use this vulnerability to achieve the same ends. Alternatively, exploits may also leverage different vulnerabilities to produce the same unwanted behavior. Below, we enumerate the vulnerabilities and information assets commonly leveraged by attacks and identify their sources.

2.3.1 Vulnerabilities

Vulnerabilities are transitions in the concrete implementation that move the machine to an unintended state. By leveraging a vulnerability, an attacker moves the implementation FSM to a state that the programmer did not consider, enabling incorrect behavior or the evasion of security protections not represented within the present state. These dangerous transitions originate from all phases of the program life cycle, from programmer misconceptions to the physical hardware's subtleties. We enumerate classes of common vulnerabilities in Table 2.1, categorizing them by their ***source***, or the mechanism that introduces the offending transition into the FSM at the end of that phase. Below, we detail the vulnerabilities introduced during each phase of our framework.

Vulnerabilities originating from high-level programming are often triggered by program inputs (*e.g.,* arguments passed through the command line), whereas vulnerabilities from subsequent phases are triggered by other influenceable states (*e.g.,* branch predictor state). An attacker can trigger these transitions to shift the implementation FSM beyond the intended

specification by carefully interfering with the target processor. However, to do so, an attacker must be able to successfully find and trigger a vulnerability. When this vulnerability originates from high-level programming, an attacker must have intimate knowledge of the program source code to locate this single fault. While this appears challenging, the commonality of some program-level vulnerabilities, like buffer overflows, eases this task. Additionally, vulnerabilities originating from hardware realization will be shared across multiple programs running on the same target platform. As a result, hardware-based vulnerabilities can be more pervasive and harder to address due to the fixed nature of physical processors.

Publicly disclosed cybersecurity flaws are assigned a unique Common Vulnerabilities and Exposures (CVE) identifier, which pertains to a *particular* vulnerability instance leveraged for an attack. These CVEs are related to one or more vulnerability classes, which are categorized via the Common Weakness Enumeration (CWE).[2] For example, the official reference to Meltdown [17] is CVE-2017-5754, which is an instance of the class CWE-200: *Exposure of Sensitive Information to an Unauthorized Actor*. CVE-2019-3568 documents a recent buffer overflow vulnerability in the WhatsApp Voice over IP (VoIP) stack, attributing it to CWE-122: *Heap-based Buffer Overflow* and CWE-119: *Improper Restriction of Operations within the Bounds of a Memory Buffer*. Due to the complex and intersecting nature of CWEs, we adopt language from the security literature to classify vulnerabilities. Table 2.1 uses this language when categorizing vulnerabilities and lists examples of CWEs belonging to each superclass.

2.3.1.1 Vulnerabilities Originating From High-Level Programming

During high-level programming, language-level undefined semantics introduce vulnerabilities in the form of non-deterministic edges that are later defined during compilation or hardware realization. Language-level undefined semantics leveraged by the security exploits covered in this work include ***spatial and temporal memory access errors*** (`V-1`, `V-2`), the use of ***uninitialized values*** (`V-3`), and ***signed integer overflow*** (`V-4`). We note that these vulnerabilities may also persist in the source code due to programmer misconceptions about the functionality of these operations. For example, a buffer overflow is a language-level undefined semantic, but it may also exist in the program due to a flawed notional computer model (*e.g.,* a programmer that is unaware that C uses zero-based indexing, as discussed in Section 2.2.1). Misconceptions and undefined semantics are often intertwined, working concurrently to produce flaws in the source code. Misconceptions can also introduce vul-

[2]The CVE and CWE lists are maintained by The MITRE Corporation with sponsorship from the U.S. Department of Homeland Security (DHS) and the Cybersecurity and Infrastructure Security Agency (CISA).

Vulnerability	Phase	Source	Unintended FSM Transition
1. Spatial Memory Access Errors *Buffer overflow, out-of-bounds access, etc.* • *CWE-121: Stack-based Buffer Overflow* • *CWE-122: Heap-based Buffer Overflow* • *CWE-787: Out-of-bounds Write* • *CWE-125: Out-of-bounds Read*	High-Level Programming	Language-Level Undefined Semantics	Based on illegal memory access
2. Temporal Memory Access Errors *Dangling pointer, use-after-free, etc.* • *CWE-416: Use After Free* • *CWE-415: Double Free* • *CWE-825: Expired Pointer Dereference*	High-Level Programming	Language-Level Undefined Semantics	Based on illegal memory access
3. Uninitialized Values *Using a value not yet assigned* • *CWE-457: Use of Uninitialized Variable* • *CWE-908: Use of Uninitialized Resource*	High-Level Programming	Language-Level Undefined Semantics	Based on unpredictable memory value
4. Signed Integer Overflow *A numeric value that exceeds the number of storage bits* • *CWE-190: Integer Overflow or Wraparound*	High-Level Programming	Language-Level Undefined Semantics	Based on modulo wrapping properties
5. Unsigned Integer Overflow *A numeric value that exceeds the number of storage bits* • *CWE-190: Integer Overflow or Wraparound*	High-Level Programming	Notional Computer Model	Based on modulo wrapping properties
6. Sharing-Dependent Program Execution *Program execution that depends on shared state* • *CWE-1303: Non-Transparent Sharing of Microarch. Resources*	Hardware Realization	Physical Processor Model	Based on externally influenced state
7. Sharing-Dependent Program Timing *Program timing that depends on shared state* • *CWE-1303: Non-Transparent Sharing of Microarch. Resources*	Hardware Realization	Physical Processor Model	Based on externally influenced state
8. Mispeculation* *Unintended execution of spurious instructions* • *CWE-1342: Info. Exposure through Microarch. State after Transient Exec.*	Hardware Realization	Physical Processor Model	Based on (incorrect) speculative state
9. Side Channels *Confidential information propagated to unintended outlets* • *CWE-203: Observable Discrepancy* • *CWE-200: Exposure of Sensitive Info. to an Unauthorized Actor*	Hardware Realization	Physical Processor Model	Reveals secret state information
10. Usage-Dependent Analog Behavior *Rowhammer, A2 attack [50], etc.* • *CWE-1255: Comparison Logic is Vuln. to Power Side-Channel Attacks* • *CWE-1300: Improper Protection of Physical Side Channels*	Hardware Realization	Physical Processor Model	Based on auxiliary state

Table 2.1: **Enumeration of Vulnerabilities.** Vulnerabilities are transitions to unintended states in the FSM. We classify vulnerabilities by their phase and source and provide examples of CWEs (child classes) belonging to each vulnerability class. In the text, we refer to this enumeration of vulnerabilities using the notation `V-1`. This list is not exhaustive of all vulnerabilities, but rather represents those most prevalent in security exploits. *The CWE enumeration can be slow to categorize emergent attacks. As a result, exploits that leverage mispeculation, such as LVI [4] (CVE-2020-0551), are often listed as having no associated CWE (written as NVD-CWE-noinfo).

nerabilities when undefined semantics are not in play. For example, ***unsigned integer overflows*** (V-5) can be just as dangerous and prevalent as signed integer overflows despite being fully defined by the C/C++ specification. Because these vulnerabilities are introduced during high-level programming, their expressiveness depends heavily on the remaining implementation. For example, the layout of the stack and presence of the NX-bit [51] will limit the viability of a buffer overflow vulnerability. In Section 2.5.1, we discuss how control-flow attacks heavily use language-level vulnerabilities to subvert program constraints.

2.3.1.2 Vulnerabilities Originating From Compilation

Compilation can also introduce transitions to unintended states into the resultant instruction-level FSM. These effects include optimizations that modify the timing or memory access patterns of programs, as discussed in Section 2.2.2.1. Changes made during compilation may also reintroduce program-level vulnerabilities. For example, consider a program that computes the size of an integer array by multiplying the number of array elements by `sizeof(int)` and stores this result in an integer variable called `sz`. If the number of array elements is sufficiently large, the variable `sz` will overflow because it is only represented by 32 bits. To prevent this signed integer overflow vulnerability, the program checks if `sz` is smaller than the number of array elements and aborts to indicate this exception. Since a signed integer overflow is undefined in the C language, the compiler may assume that the if-condition is always false (*i.e.,* `sz` cannot be smaller than the number or array elements) and mark it as dead code [36]. Thus, this flaw in the compiler's abstract processor model permits the removal of this security check, resulting in code that contains an integer overflow vulnerability despite the programmer's intent to actively avoid one. With respect to our framework, although this compiler operation removes a security check, it introduces unintended states and transitions into the resultant FSM in which `sz` is smaller than the number of array elements and the program counter continues to advance. Such an integer overflow vulnerability could be used to allocate a zero-sized array or hash table to enable arbitrary code execution, as was done by attacks on SSH1 [52].

2.3.1.3 Vulnerabilities Originating From Hardware Realization

Vulnerabilities introduced during hardware realization originate from the physical processor model and especially from design choices related to shared and auxiliary state. Specifically, the physical processor model adds transitions dependent on influenceable states (*i.e.,* shared state or writable auxiliary state) or forms paths that propagate confidential information to observable outlets (*i.e.,* shared state or readable auxiliary state). These formulations are un-

intended with respect to the intended FSM and pervasive across architecture-based attacks. For example, ***mispeculation*** (V-8) permits the execution of unintended instructions. In isolation, mispeculation is not dangerous, as discussed in Section 2.2.3. But, when combined with vulnerabilities originating from hardware realization, transient execution becomes a powerful tool to perpetrate security exploits. Attackers can influence transient execution due to ***sharing-dependent program execution*** (V-6) in the microarchitecture. Specifically, shared state, like the branch predictor contents, affects program transitions, enabling an attacker to control execution without supplying program input directly. Furthermore, transient execution leaves residues in the cache, affecting shared state (cache contents) and readable auxiliary state (timing), due to the presence of a ***side channel*** (V-9) vulnerability. With these two mechanisms in hand, an attacker can force a program to execute an instruction speculatively and observe the traces of that execution. In addition to sharing-dependent program execution and side channels, we recognize ***usage-dependent analog behavior*** (V-10) as a vulnerability introduced during hardware realization. This vulnerability captures transitions caused by analog-based auxiliary state (Section 2.2.3.3), such as how a bit flip can occur in DRAM due to repeated accesses to a neighboring row [40].

2.3.2 Implementation Information Assets

Knowledge of the state space of the implementation FSM, termed implementation information, is leveraged directly for security exploits. Given this information, an attacker has a better understanding of vulnerability sources and both unintended states and transitions in this FSM. Hence, the attacker has a stronger and more refined model of the semantics of the implementation than the original programmer. Attackers can take advantage of this information to find a way to synthesize an exploit and subvert the program's security measures. It should be noted that implementation information does not detail specific states or transitions, but is a property of the state space as a whole. This knowledge is, in effect, more compelling, as an attacker can often successfully synthesize exploits with only a few fundamental information assets. Table 2.2 enumerates information assets that are typically required by exploits. We categorize information assets by their *source*, or the mechanism that introduces the state information during the program life cycle. We also categorize information assets by the method in which knowledge of it is acquired, termed *Implementation Information Acquisition*, through one of the following channels: documentation (DOC), reverse engineering (ENGR), or probing (PROBE). These methods of acquisition broadly capture three possible scenarios for understanding a particular system aspect: *i)* The system aspect is documented publicly and is known by the attacker or can be obtained easily, *ii)*

The system aspect is not documented publicly but can be directly derived using knowledge of other system aspects, or *iii)* The system aspect can only be indirectly derived by querying the system and observing a response. Below, we discuss implementation information acquisition in depth, and enumerate information sources in each stage of our framework.

Implementation Information Asset	Phase	Source	Acquisition
1. Code Contents. *Behavior of functions, etc.*	High-Level Programming	Notional Computer Model	DOC, ENGR, PROBE
2. Location of Code. *Absolute location of functions, code gadgets etc.*	Compilation	Compiler Choices*	ENGR
3. Order of Code. *Relative location of functions, code gadgets etc.*	Compilation	Compiler Choices	DOC, ENGR
4. Location of Data. *Absolute location of stack frames, heap allocations, etc.*	Compilation	Compiler Choices*	ENGR
5. Order of Data. *Order of variables within a stack frame, etc.*	Compilation	Compiler Choices	DOC, ENGR
6. Representation of Code Pointers. *Binary representation of code addresses*	Compilation	Compiler Choices	DOC
7. Representation of Data Pointers. *Binary representation of data addresses*	Compilation	Compiler Choices	DOC
8. Representation of Code. *Binary representation of instructions*	Compilation	Compiler Choices	DOC
9. Representation of Data. *Binary representation of data values*	Compilation	Compiler Choices	DOC
10. Microarchitectural Sharing. *Cache collisions, shared BTB entries, etc.*	Hardware Realization	Microarch. Choices	ENGR, PROBE
11. Timing of Operations. *Latency of memory accesses, etc.*	Hardware Realization	Microarch. Choices	PROBE

Table 2.2: **Enumeration of Implementation Information Assets.** We classify information assets by their phase and source, whether it is introduced during high-level programming via the notional computer model, during compilation via compiler choices, or during hardware realization via microarchitectural choices. Furthermore, we specify how this implementation information is acquired by attackers via documentation (DOC), reverse engineering efforts (ENGR), or probing techniques (PROBE). In the text, we refer to this enumeration of information assets using the notation `I-1`. This list is not exhaustive of all implementation information assets, but rather represents the classes most prevalent in security exploits. *Location of Objects originates from OS Choices when ASLR [5] is in place.

2.3.2.1 Implementation Information Acquisition

Security exploits require some knowledge of the implementation information to succeed. Control-flow attacks that exploit program-level vulnerabilities typically require knowledge of the memory layout, stack organization, and code locations. Similarly, hardware-based

attacks, like timing side channels, require knowledge of microarchitectural structures and sharing between processes. This knowledge is obtained using acquisition methods (DOC, ENGR, or PROBE) at each level of the program life cycle.

By leveraging documentation (DOC) and reverse engineering techniques (ENGR), an attacker can gain knowledge of system aspects determined during high-level programming and compilation. Documentation and formal specifications of the programming-language and operating system are among the most trivial ways to obtain knowledge of these system aspects. For example, attackers can leverage software documentation, open-source code, or knowledge of popular libraries to understand program functionality and code contents.[3] Alternatively, reverse engineering the program binary can also reveal implementation decisions made during compilation. For example, disassembling the program binary can reveal implementation decisions made during compilation (*e.g.*, call stack organization), and information about the code addresses (*e.g.*, absolute or relative code locations, depending on the underlying runtime environment).

Finally, attackers may attempt to glean information assets by probing the system via sophisticated program analysis and side channels (PROBE). These experiments repeatedly measure physical phenomena like timing to identify correlations and infer properties of the underlying system. For example, timing memory accesses for a repeated set of array lookups can reveal cache associativity. Such implementation information is required by popular timing side-channel attacks like PRIME+PROBE [53]. To derive the cache layout, PRIME+PROBE selects a potential conflict set from a large buffer, then repeatedly *probes* if this set evicts a candidate memory line from the LLC by measuring memory access times (*i.e.,* readable auxiliary state). Candidate memory lines are added to the conflict set if the memory access is fast, implying that they were not evicted. Once the conflict set is formed, the attack repeats a similar process to partition the conflict set into an eviction set for a specific candidate line. After this second iteration, the attacker has acquired sufficient knowledge of microarchitectural sharing in the LLC to synthesize the side-channel exploit. Given that processor design is proprietary, probing experiments present the best mechanism to learn information about the hardware. Additionally, because this acquisition mechanism exploits physical phenomena, it is both hard to anticipate a priori and hard to combat.

If the implementation is totally opaque to the user, security exploits cannot prevail. However, information leakage is innate to many implementation decisions by design, therefore, opaque implementations are not a universal solution. For example, the timing of a memory access reveals if there is a hit or miss in the cache. This property is inherent to the cache

[3]We are fully supportive of the open source movement, but want to point out this aspect of obtaining information assets.

design, yet leaks information that can be used to determine the contents of memory. Rather than making implementation information completely opaque, an alternative solution is to make it unreliable and inconsistent. Such randomization techniques increase the difficulty of implementation information acquisition. For example, ASLR [5] determines memory layout at load-time, forcing attackers to obtain memory information using sophisticated probing techniques when only reverse engineering was required previously [23, 24, 25, 26]. Randomization defenses are further characterized in Section 2.4.3.

2.3.2.2 Information Assets From High-Level Programming

Perhaps the most critical information assets required for exploits is knowledge of the ***contents of the victim source code*** (I-1). Code contents detail how the intended FSM is implemented as source code, capturing program functionality like function contents, input formatting, and operational behaviors. All security attacks exploit some degree of knowledge of a victim program. Stack smashing attacks must understand how to provide program input that triggers a buffer-overflow vulnerability [54]. Code reuse attacks must know the functionality of code gadgets [37, 55]. Timing side-channel exploits rely on secret-dependent program execution (*e.g.,* key-dependent conditional logic) [39]. Spectre attacks require knowledge of a Spectre gadget, or jump instruction in the victim program [16]. To acquire this information, attackers may use documentation, reverse engineering, or probing techniques. Attackers leverage software documentation, open-source code, or knowledge of popular libraries to discover program functionality easily (DOC). If the source code is not widely available but the binary is, disassembling the binary can provide similar information (ENGR). If neither is available, attackers may attempt to glean program behavior through probing the API, repeatedly sending inputs to profile the results (PROBE).

2.3.2.3 Information Assets From Compilation

The compiler implements many components of the implementation FSM that are critical to program execution, including the ***location and order of objects*** (I-2—I-5). For example, during compilation, the operating system, compiler, and target instruction-set dictate the implementation of function calls in the source code. This is traditionally done through the manipulation of a call stack. Values stored to the stack record the control flow of the program and pass arguments to the requested function. In addition, the call itself decomposes into multiple instructions that calculate the target function pointer and jump to the resulting address. These abstractions are hidden from the programmer in high-level languages and add state to the resulting implementation that is critical for stack smashing exploits [54].

In addition to data and code layout, the compiler also implements the ***representation of code and pointers*** (I-6—I-8). While these mechanisms are heavily constrained by what the target processor supports, they are introduced into the resultant binary and instruction-level FSM during compilation.

2.3.2.4 Information Assets From Hardware Realization

Hardware realization massively expands the state space of the implementation FSM based on microarchitectural choices. The resultant machine contains shared and auxiliary state, including properties of ***microarchitectural sharing*** (I-10) and the ***timing of operations*** (I-11). Knowledge of this implementation information is leveraged directly for transient execution attacks (Section 2.5.4), like Spectre V2 [16]. Spectre attacks leak information by combining speculative execution with data exfiltration through timing side channels. Spectre V2 does this by mistraining the processor's branch predictor to mispredict a specific indirect branch of the attacker's choosing, then observing the residues of this transient execution through the cache. While this attack requires some knowledge of program-level information, like the location of the victim branch, it also requires intimate knowledge of the branch predictor. To reliably mistrain the predictor, the attackers used experiments to reverse engineer (ENGR) the branch predictor on the target machine. They found that a predictor on a Haswell i7-4650U only stores the lower 20 bits of the virtual address for twenty-nine previous destination addresses [16]. Using this knowledge, they were able to craft malicious code to mistrain the predictor reliably. The remainder of the attack recovers cache residues by performing FLUSH+RELOAD [56].

Defenses may also seek to instantiate implementation information during hardware realization to prevent attacks. Address Space Layout Randomization (ASLR) [5] randomizes the location of the code segment when the program is loaded. Hence, function pointers will vary per instance of the program. ASLR adds diversity to the state space of the implementation as each instance of the program now has a unique memory configuration. With memory locations changing at runtime, obtaining information assets requires advanced leakage techniques that are much more difficult than if memory locations were determined at compile time.

2.3.3 Summary of Security Exploits

Security exploits leverage a vulnerability to transition the implementation FSM to an unintended state and bypass the programmer's intent, then use knowledge of implementation information assets to manipulate the implementation FSM and produce unexpected and un-

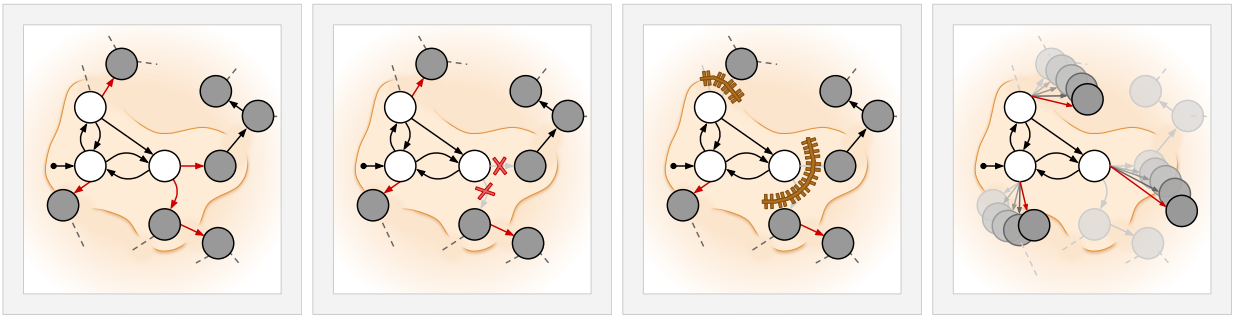
wanted behavior. With respect to our framework, we enumerate and classify vulnerabilities by their source, identifying prominent vulnerabilities originating from language-level undefined semantics, the notional computer model, and the physical processor model, shown in Table 2.1. We similarly enumerate and classify information assets originating from compiler and microarchitectural choices in Table 2.2. We also characterize how attackers acquire knowledge of implementation information needed for exploits, identifying three primary methods with increasing levels of sophistication: documentation, reverse engineering, and probing. In Section 2.5, we discuss how prominent attacks fit within our model of security exploits.

2.4 Classification of Defenses

Exploits subvert the security guarantees of an application by using a vulnerability to transition to an unintended state and then using knowledge of implementation information assets to produce unexpected behavior. To combat attacks, defenses address the usage of unintended states by either preventing the transition to an unintended state (*i.e.*, via removing a vulnerability), or preventing the subversion of security guarantees (*i.e.*, via obfuscating the state space), effectively making it impossible or useless to acquire unintended states or transitions. Namely, protections either ***avoid*** vulnerabilities by removing their instances in applications, ***enforce*** a property of the machine to mitigate vulnerabilities, or ***obfuscate*** information assets to make the implementation unpredictable and thus difficult to manipulate. These categories are not mutually exclusive, as some defenses have included both enforcement and obfuscation-based techniques in their designs.

In addition to classifying defenses by type, we specify their level of sophistication, quantified by the ***strength*** or coverage of the defense and the ***phase*** during which it is employed. Strength captures the completeness to which a vulnerability is protected or, for obfuscation-based defenses, the entropy[4] of the system. Phase captures when a defense applies in the program's life cycle. For defenses introduced during hardware realization, we distinguish between *load-time* and *runtime* defenses. Namely, defenses instantiated at load-time consistently change the state space but do not effect execution. For example, ASLR [5] is a load-time defense since the program offsets are constant for each execution. Runtime defenses take on multiple values during program execution. Runtime defenses include ASLR variants that re-randomize the program offset during execution [12, 31, 57, 58, 59]. Due

[4]In this work, we define entropy as the number of key bits used to obfuscate unintended state. We denote obfuscation techniques as having low entropy if they use key sizes smaller than 32-bits or employ weak XOR ciphers.



(a) **Unprotected.** (b) **Avoidance.** (c) **Enforcement.** (d) **Obfuscation.**

Figure 2.2: **Defense Techniques.** Vulnerabilities are exploited to transition the victim program from an intended state (white) to an unintended state (grey). Once outside of the constraints of the intended IFSM, an attacker can use knowledge of information assets to subvert security. Avoidance-based protections (2.2b) find and fix vulnerabilities to remove single transitions to unintended states. Enforcement-based protections (2.2c) take a complete approach by addressing *all* transitions of a particular type (*e.g.*, buffer overflows). Finally, obfuscation techniques (2.2d) conceal information assets and unintended states, preventing attackers from effectively and reliably learning enough information to synthesize an exploit.

to differing sophistication, two defenses that address the same class of unintended states or transitions in the same manner may not be equivalent. Below, we detail these three defense techniques in more detail.

2.4.1 Avoidance of Vulnerabilities

Designers use testing or formal verification to avoid unintended states and ensure that these states are not reachable from the program-level. Figure 2.2a illustrates an unprotected system, where unintended transitions are leveraged to shift the program from an intended state, shown in white, to an unintended state, shown in grey. ***Avoidance techniques***, Figure 2.2b, remove single vulnerabilities, eliminating *specific* unintended transitions in the implementation FSM.

For example, a program with a buffer overflow vulnerability will have a transition to a state where memory has been corrupted. Avoidance prevents an exploit by removing the single buffer overflow vulnerability (*i.e.*, the single transition) to the corrupt state, through program testing. However, program testing is not exhaustive. Avoidance does not address the existence of spatial memory access errors as a whole. Other vulnerabilities may still exist, allowing for exploits that abuse vulnerabilities elsewhere in the source code. Rather than relying on program testing, programmers may employ formal verification to eliminate vulnerabilities. While formal verification is comprehensive, it currently does not scale to

large programs. Additionally, avoidance defenses are only a durable protection until someone writes more code and introduces a new vulnerability. Lastly, since testing and formal verification techniques apply to source code or binaries, the avoidance approach cannot be easily extended to address vulnerabilities in the microarchitecture.

2.4.2 Enforcement of Program Semantics

Enforcement protections seek to remove all vulnerabilities of a particular type from an application by prohibiting the existence of a vulnerability altogether. To eliminate vulnerabilities, enforcement protections detect any uses of the vulnerability, sometimes effectively defining an undefined semantic (*i.e.*, marking it as erroneous). With respect to Figure 2.2c, enforcement creates barriers around parts of the implementation FSM. Rather than removing a single transition, as in the case of avoidance, enforcement provides significant obstacles to initiating the security exploits by prohibiting *all* unintended transitions of some type.

2.4.2.1 Enforcement During High-Level Programming

At the programming language level, enforcement works to remove all instances of undefined semantics by fully defining these semantics within the programming language specification. This category includes memory-safe languages, like Java, that eliminate spatial and temporal memory errors. Program-level enforcement techniques are effective but may not be comprehensive. These techniques cannot secure microarchitectural assets; therefore, they are incomplete due to the nature of the underlying hardware. Additionally, these languages are sometimes implemented in an unsafe programming language, leaving possibilities for exploits. While fully-defined programming languages are effective, we choose to focus on defenses that secure C or C++ programs in our study, since these programming languages are popular and used in many legacy applications. We assume it is less practical to re-implement these applications in a safe programming language, but we do acknowledge that this possibility exists.

2.4.2.2 Enforcement During Hardware Realization.

Runtime enforcement protections prohibit vulnerabilities and undefined semantics by detecting their use during program execution. For example, processors like CHERI [60] enforce spatial memory access errors at runtime by triggering a security exception whenever these vulnerabilities are detected. Alternatively, Control-Flow Integrity (CFI) [61] enforces memory access errors that overwrite jump targets by validating all control transfers against the intended FSM. CFI is complete with respect to its threat model, but incurs high performance

overheads, motivating coarse-grained variants that proved to be susceptible to attackers [62]. Ultimately, once attacks evolved to no longer rely on the protected asset, enforcement defenses were easily bypassed. Our case-study in Section 2.5.1 shows evidence of this trend.

2.4.3 Obfuscation of Implementation Information Assets

Finally, some protections obfuscate implementation information assets to make knowledge of the implementation FSM unreliable, either by making it difficult to acquire knowledge of the implementation or rendering this knowledge useless. With respect to Figure 2.2d, an obfuscation defense conceals or obscures unintended states, increasing the sophistication of information acquisition needed by attackers to synthesize an exploit. Obfuscation defenses mitigate attacks by making the information assets difficult (albeit not impossible) to acquire by making unintended states unpredictable. Our studies revealed that a majority of attacks require at least some knowledge of information assets. Thus, obfuscation-based defenses are a promising method to mitigate a range of security attacks.

2.4.3.1 Obfuscation During Compilation

Compile-time defenses randomly instantiate a parameter during program compilation. Thus, this parameter has the same value every time a program runs. However, the same program on another machine may have a different instantiation. These compiler based protections are used most commonly for binary obfuscation. For example, Polyverse [63] provides a compiler that randomizes register usage, function locations, import tables, and other targets per compilation. Namely, each binary produced by Polyverse is unique while preserving the semantics of the intended program. This tool effectively mitigates control-flow exploits by obfuscating critical attack assets, like code location. Compiler-level obfuscation techniques apply one-time randomization. Thus, although the implementation information is initially unknown, it can be easily discovered by attackers as this information is constant for each instance of the program. Therefore, these defenses are the most susceptible to information leakage and derandomization attacks (Section 2.5.2). As the time of obfuscation is moved closer to program execution, discovering a randomized parameter becomes much more difficult.

2.4.3.2 Obfuscation During Hardware Realization

Defenses during hardware realization randomly instantiate an undefined semantic for each run of a program. Thus, any knowledge the attacker gains about the implementation of a obfuscated semantic cannot be used to attack another instance of that program. The most

common example of load-time randomization defenses is Address Space Layout Randomization (ASLR) [5]. Many control-flow attacks rely on knowledge of code locations. Finding the address of a target function can be done trivially by disassembling the binary. Yet, with ASLR [5], which randomizes the location of objects at load time, the address of a target function changes per execution. Disassembly no longer reveals the true code location, making the attack more difficult. Unfortunately, clever probing of shared state can still defeat most one-time randomization defenses, including 64-bit ASLR, since the randomized information is constant at runtime [23, 24, 25, 26]. These derandomization attacks are discussed in Section 2.5.2.

2.4.3.3 Obfuscation During Runtime Execution

Defenses that obfuscate assets at *runtime* have appeared in response to derandomization attacks. Within these defenses, the protected asset (*e.g.,* data location) takes many forms during a single execution of the program. While these defenses are still instantiated during hardware realization, we distinguish this from the prior section because the value of the randomized information changes multiple times during the program's lifetime. For example, Timely Address Space Randomization (TASR) [57] relocates the code segment during runtime after every system call, effectively obfuscating the location of code objects. This defense aims to mitigate memory leaks that derandomized traditional ASLR by changing code location immediately after a leak could occur (via a system call). Other runtime obfuscation defenses change assets at a constant rate, such as every 50 ms, rather than in response to another event. Generally, these defenses employ higher entropy and more frequent obfuscation schemes to create a tight time limit that probing techniques must overcome to be successful [12, 31, 59].

2.4.4 Summary of Defenses

Defenses mitigate security exploits in three primary ways: *i)* Avoidance of vulnerabilities, *ii)* Enforcement of the implementation FSM to prevent vulnerabilities from being leveraged for attacks, and *iii)* Obfuscation of information assets to prevent attackers from reliably exploiting the implementation FSM. These approaches are illustrated in Figure 2.2. With respect to our framework, we classify defenses by the phase during which it is employed, making a distinction between load-time and runtime defenses within hardware realization, and their coverage of vulnerability sources. In Section 2.5, we discuss how prominent mitigation techniques fit within this model.

2.5 Case Studies of Attack Genealogies

In Section 2.2, we showed how the program life cycle introduces vulnerabilities in its different phases. Following this, Section 2.3 went deeper into how exploits use these vulnerabilities by obtaining implementation information that can then be used to transition the implementation FSM to unintended states. Section 2.4 then outlined categories of defenses in terms of how they prevented the attacker from gaining control through transitioning to these unintended states.

In this section, we study four prominent attack genealogies: ***control-flow attacks***, ***derandomization attacks***, ***timing side channels***, and ***transient execution attacks***. These case studies illustrate how prominent attacks and defenses fit into our framework and serves as background for the remaining chapters in this dissertation. The first two genealogies compromise program integrity by hijacking the victim's control flow or, in the case of derandomization attacks, do so after subverting a randomization-based defense. The later two genealogies compromise program confidentiality by using microarchitectural vulnerabilities to leak sensitive information.[5] For each case study, we provide *i)* An enumeration of prominent security exploits and the vulnerability sources they leverage, as related to our program life cycle framework through Tables 2.1 and 2.2, and *ii)* An enumeration of defenses classified by their approach (Figure 2.2) and the vulnerability sources they protect. These case studies serve to illustrate:

- How the attacks and defenses for each case study fit into the framework of this chapter.
- How the attacks have evolved over time with increasingly sophisticated acquisition of information assets by the attackers and the mechanisms used to acquire them.
- How the defenses have evolved over time with an increasing shift from avoidance to enforcement and obfuscation—techniques which eliminate access to large parts of the unintended state space and thus are potentially more durable.

2.5.1 Control-Flow Attacks

Control-flow attacks are characterized by attempts to redirect program execution from paths intended by the programmer. We adopt the following definition:

> ***Contol-Flow Attack:*** A class of security exploits that redirect the flow of execution (*i.e.*, the control flow) of a target program. This goal is often achieved through manipulating

[5] While we do not consider availability attacks within our case studies, our framework could be adjusted to capture these exploits, as discussed in Section 2.6.

return addresses, code pointers, loop variables, or conditions. Control-flow attacks sometimes leverage code injection or redirect execution to existing code on the target device. This category of attacks is also known as control-flow hijacks or arbitrary code execution (ACE) attacks. We use these terms interchangeably. ACE attacks that occur over the network are often referred to as remote code execution (RCE) attacks. Code injection is a subset of control-flow attacks.

Attacks that manipulate data or data pointers to redirect control flow (*e.g.*, Data-Oriented Programming [64]) are categorized as "control-flow attacks." In this work, we distinguish between two classes of control-flow attacks: those that occur on the ***control plane*** (*i.e.*, require manipulation of code or code pointers) and those that occur on the ***data plane*** (*i.e.*, require manipulation of data or data pointers). This distinction is relevant to Chapters 4 and 5 of this dissertation, as the defenses presented in these chapters only guard against control-flow attacks on the control plane.

Control-flow attacks abuse a memory access error to transition the implementation FSM to an unintended state, traditionally this state being one that permits unauthorized access to the computer. Specifically, these exploits abuse a memory access error to overwrite control data (*e.g.,* a return address or code pointer) with a target of their choosing. To do this reliably, attackers must carefully craft an attack payload using knowledge of information assets. Specifically, all control-flow attacks require knowledge of the data layout in order to accurately corrupt the intended information. Additional implementation information that is needed is dependent on the chosen target; code injection necessitates knowledge of the representation of code pointers and code, whereas code reuse necessitates the location of a target function. We provide a chronological listing of attacks that have leveraged novel vulnerability sources in Table 2.3 and prominent control-flow attacks and defenses in Table 2.4. Below, we detail how these attacks have evolved, from early work focused on exploiting the stack, to advanced attacks that leveraged existing code and non-control data to redirect execution.

2.5.1.1 Code Injection

The entire class of control-flow exploits originated from the first stack-smashing attack in the late 1990s [54]. Within this exploit, an attacker leverages a buffer overflow vulnerability on the stack to influence unintended state in two ways: to inject malicious code into the data segment, and to overwrite the return address with a pointer to the injected code. The target processor proceeds from this corrupt state, popping the return address and executing the malicious code to conclude the exploit. To craft a payload to synthesize this exploit

	Attacks	Vulnerability/Info. Asset	Source	Acquisition
1996	**Stack Smashing/Buffer Overflow [54]**	MAE, *Overwrite Code Ptr.* (V-1)	Lang-Level U.S.	–
		Order of Data, *Stack* (I-5)	Compiler Choices	DOC, ENGR
		Location of Data, *Stack* (I-4)	Compiler Choices	ENGR
		Representation of Code (I-8)	Compiler Choices	DOC
		Representation of Code Pointers (I-6)	Compiler Choices	DOC
1997	**Return-into-libc** [37]	Location of Code (I-2)	Compiler Choices	ENGR
1999	**Heap Smashing/Buffer Overflow [65]**	Order of Data, *Heap* (I-5)	Compiler Choices	DOC, ENGR
2001	**Heap Spray [66]**	Location of Data, *Heap* (I-4)	Compiler Choices	ENGR
2016	**Data Oriented Programming [64]**	MAE, *Overwrite Data* (V-1)	Lang-Level U.S.	–
		Representation of Data Pointers (I-7)	Compiler Choices	DOC
		Representation of Data (I-9)	Compiler Choices	DOC

Table 2.3: **Novel Vulnerability Sources in Control-Flow Attacks.** We enumerate control-flow attacks that have leveraged a novel vulnerability or information asset (*i.e.,* one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition. For space driven brevity, we adopt the following abbreviations: memory access error (MAE) and undefined semantics (U.S.).

successfully, the attacker requires the following knowledge of implementation information from compilation: *i)* The order of data on the stack (to overflow the buffer), *ii)* The representation of code (to inject code), *iii)* The location of data on the stack (to redirect to injected code), and *iv)* The representation of code pointers (to overwrite the return address). Additionally, unintended states in the implementation FSM must permit the execution of stack data. Attackers can derive knowledge of this information assets from documentation of the system and target processor.

Two novel protections were promptly adopted to prevent stack smashing attacks. In 1997, the non-executable stack patch forbid the execution of data on the stack, effectively enforcing intended states in the implementation FSM after a memory access error occurs [70]. A year later, stack canaries leveraged obfuscation to detect linear buffer overflows by asserting that a secret value on the stack had not been corrupted during runtime [71]. While these prominent defenses protected the call stack, they were not comprehensive of all memory. Attackers quickly subverted these defenses by exploiting buffer overflows and performing code injection within the heap [65, 66]. Shortly afterward, the protections of the non-executable stack expanded to all data to fully eradicate code injection. Specifically, Intel introduced the no-execute bit (NX-bit) [51] to its x86 instruction-set architecture to forbid data execution, effectively preventing these unintended states from being entered due to memory corruption. In response, attackers adapted their techniques to exploit other unintended states and transitions still present on the target machine.

			MAE for CP	MAE for D	Code Contents	Order of Data	Loc. of Data	Loc. of Code	Rep. of Code	Rep. of CP	Rep. of DP	Rep. of D
	Attacks											
1996	**Stack Smashing (Stack Buffer Overflow) [54]**		●	○	◐	◐	◐	○	●	●	○	○
1997	**Return-into-libc [37]**		●	○	◐	◐	○	●	○	●	○	○
1999	**Heap Smashing (Heap Overflows) [65]**		●	○	◐	◐	◐	○	●	●	○	○
2001	**Heap Spray [66]**		●	○	◐	◐	◐	○	●	●	○	○
2007	**Return Oriented Programming (ROP) [55]**		●	○	◐	◐	○	●	○	●	○	○
2011	**Jump Oriented Programming (JOP) [67]**		●	○	◐	◐	○	●	○	●	○	○
2014	**Call Oriented Programming [68]**		●	○	◐	◐	○	●	○	●	○	○
2015	**Control-Flow Bending [69]**		●	○	◐	◐	○	●	○	●	○	○
2016	**Data Oriented Programming [64]**		○	●	◐	◐	○	○	○	○	●	●
	Defenses											
1997	**NX-Stack [70]**	*Runtime Enforcement*	◐	○	○	○	○	○	○	○	○	○
1998	**Stack Canaries [71]**	*Load-Time Obfuscation*	◐	○	○	○	○	○	○	○	○	○
2003	**NX-Bit [51]**	*Runtime Enforcement*	◐	○	○	○	○	○	○	○	○	○
	Instruction Set Randomization (ISR) [72, 73]	*Compile-Time Obfuscation*	○	○	◐	○	○	○	●	○	○	○
	Addr. Space Layout Randomization (ASLR) [5]	*Load-Time Obfuscation*	○	○	○	○	●	●	○	○	○	○
	PointGuard [28]	*Load-Time Obfuscation*	○	○	○	○	○	○	○	●	●	○
2009	**Control Flow Integrity (CFI) [61]**	*Runtime Enforcement*	●	○	○	○	○	○	○	○	○	○
2013	**kBouncer [74], ROPecker (2014) [75]**	*Runtime Enforcement*	◐	○	○	○	○	○	○	○	○	○
2014	**Code-Pointer Integrity (CPI) [76]**	*Runtime Enforcement*	●	○	○	○	○	○	○	○	○	○
	CHERI [60]	*Runtime Enforcement*	●	●	○	○	○	○	○	○	○	○
2018	**Random Embedded Secret Tokens (REST) [77]**	*Runtime Obfuscation*	●	●	○	○	○	○	○	○	○	○
2021	**Cryptographic Capability Computing (C^3) [27]**	*Runtime Obfuscation*	◐	◐	○	○	○	○	○	○	◐	◐

Table 2.4: **Chronological Ordering of Control-Flow Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent control-flow attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: memory access error (MAE), code pointer (CP), and data pointer (DP).

2.5.1.2 Code Reuse

With the demise of code injection, exploits morphed to *reuse* existing code on the victim device. Code reuse attacks originated from return-into-libc [37], a variant of the stack smashing exploit that redirects program execution to a library function. This attack is identical to the stack smashing exploit above except that, rather than requiring knowledge of data location

and code representation, knowledge of *code location* is leveraged to redirect execution to an unintended library function. While library functions do exist in the implementation, the states used in return-into-libc are unintended because this specific stack layout and execution pattern is not permitted by the intended FSM.

A decade later, Return Oriented Programming (ROP) [55] revisited code reuse attacks, exploiting existing code at a finer granularity. Rather than executing a single library function, ROP stitched together short code gadgets to form arbitrary programs that were just as expressive as what code injection achieved. This attack required identical information assets as return-into-libc, relying heavily on *code contents* and *code location* to identify gadgets. Early enforcement-based protections profiled the call patterns of ROP attacks, triggering security exceptions when malicious execution was detected [74, 75]. However, these defenses were easily subverted by attacks that used call patterns outside of the assembled profile, for example, by executing gadgets that ended in jump instructions rather than returns [67, 68]. Other defenses attempted to fortify the code space, such as by preventing memory access errors from corrupting return addresses (discussed below). In response, ROP evolved to exploit data. Data Oriented Programming (DOP) [64] manipulates non-control data, instead of function pointers, to influence indirect control flow and produce unexpected behavior. Hence, rather than requiring knowledge of the location of code or representation of code pointers, DOP solely relies on implementation information related to data and data pointers.

2.5.1.3 Defenses Against Control-Flow Attacks

As shown in Table 2.4, defenses for control-flow attacks primarily fall into two categories: *i)* Obfuscation of critical information assets (*e.g.*, location of code/data, representation of code, and representation of code/data pointers), and *ii)* Enforcement of memory access errors. Obfuscation-based mitigations looked to fortify access to data and code locations, and the representation of pointers and code. Early obfuscation-based defenses aimed to obfuscate memory layout to prevent attackers from crafting pointers to injected code or existing code gadgets. ASLR [5] shifts the location of the stack, heap, and code in memory. Since this technique randomizes once, sophisticated implementation acquisition methods can be used to locate the obfuscated segments at runtime. This arms race led to an entire new genealogy of attacks, termed derandomization attacks, detailed in Section 2.5.2. Finer-grained randomization techniques [78, 79, 80] and protections that re-randomize at runtime [57] have been developed to combat these derandomization attempts. Obfuscation defenses also randomized the *representation of code* at either compile-time or load-time to thwart code injection and gadget discovery [72, 73, 81, 82]. Other techniques fortified code pointers by encrypting the *representation of pointers* [28].

Other defenses employ enforcement techniques to detect memory access errors and thwart control-flow attacks. Control-Flow Integrity (CFI) [61] analyzes the trusted program for all possible, programmer-intended execution paths in the control-flow graph and restricts execution to this graph during runtime, effectively preventing memory access errors by restricting the acceptable values of code pointers. Although CFI is a robust enforcement mechanism, it comes with high overheads. Thus, adopted CFI implementations only trace a subset of control-flow edges [83, 84, 85], leaving them vulnerable to subversion [62, 69, 86, 87]. Subsequent defenses continued to fortify control data. Code Pointer Integrity [76] uses enforcement to stop memory access errors from influencing code pointers by isolating pointers in a protective memory region. Control-Data Isolation [88] took a subtractive approach to achieve the same goal, removing all indirect control-flow instructions that can be influenced by user data (*i.e.,* indirect jumps, calls, and returns). Other protections like CHERI [60] and REST [77] employed runtime mechanisms to detect memory access errors quickly. Cryptographic Capability Computing (C^3) [27] similarly defends against spatial memory access errors by employing object-granular encryption for allocations on the heap. This approach enforces spacial safety indirectly by garbling the result of any heap overflows. C^3 was published concurrently with the work done in this dissertation and exemplifies broader consideration of encrypted architectures for security.

2.5.2 Derandomization Attacks

As we saw in Section 2.5.1, control-flow attacks leverage knowledge of compiler choices related to code and data layout to craft attack payloads. Mitigations have focused on *obfuscating* this critical implementation information to prevent attacks. Address Space Layout Randomization (ASLR) [5] is one of few widely adopted randomization defenses. ASLR obfuscates memory layout at load-time, changing the position of the data and code segments to hide location information. However, while ASLR is effective, sophisticated implementation acquisition methods can be used to gain knowledge of the randomized implementation at runtime and subvert the defense. Thus, exploits have prevailed in the face of ASLR by using these derandomization tactics. We adopt the following definition to classify this genealogy of attacks:

> ***Derandomization Attack:*** A class of security exploits that learn a randomized information asset to subvert a security defense. This goal is often achieved by leveraging advanced information acquisition techniques (§2.3.2.1) such as side channels.

The derandomization attack genealogy expands upon control-flow attacks to include advanced methods of information assets acquisition that evade obfuscation protections. We

provide a chronological listing of derandomization attacks that have leveraged novel vulnerability sources in Table 2.5 and prominent attacks and defenses in Table 2.6. Below, we detail how these derandomization attacks have evolved, from exploiting memory leakage to advanced probing tactics that leverage timing side channels.

2.5.2.1 Disclosing Code Location via Memory Leaks

Attacks responded to the adoption of ASLR by derandomizing code location before synthesizing an exploit. Initial implementations of ASLR on 32-bit systems had low entropy and were trivially defeated by control-flow attacks that brute-force guessed the randomized memory layout [89]. With the deployment of ASLR on 64-bit systems, attacks needed more sophisticated derandomization methods to overcome the increased entropy. Exploits began to use the limited coverage of ASLR to subvert the defense. In particular, while ASLR randomizes the location of objects, it does not randomize the distance between objects in a particular segment. Thus, given a leaked pointer and knowledge of *relative addressing*, attackers can easily derandomize the address space. Variants of ROP integrated these derandomization techniques to subvert ASLR prior to an attack. JIT-ROP [23] uses a memory disclosure vulnerability to leak the *absolute* location of a code object, then uses this address to read code pages and follow direct jumps to discover other parts of the code segment. Once enough of the code contents and location is known, an ROP payload is synthesized using just-in-time compilation. In addition to the assets leveraged by ROP [55] (discussed in Table 2.4), JIT-ROP uses knowledge of the *representation of code* to read code pages in order to uncover the locations of code gadgets at runtime.

To mitigate JIT-ROP, Oxymoron obfuscated direct branch targets to prevent code pages from leaking code locations [90]. Attacks that used indirect branch targets to infer code locations bypassed this protection [91]. Subsequent work extended randomization to all jump targets [91], adding a layer of indirection between code pointers and code location to confuse attackers. Other JIT-ROP mitigations worked to make code pages unreadable by enforcing different permissions for code pointers and data pointers (*e.g.,* load/store operations can only use data pointers) [92]. These randomization and enforcement techniques have been combined in Readactor [93] to efficiently mitigate direct and indirect code page discovery.

With the rise of new protections, attacks turned to more advanced acquisition techniques to learn code locations. Crash-Resistant Oriented Programming [94] demonstrated techniques to disclose code pointers without crashing the victim process, allowing this attack to sidestep enforcement protections that triggered exceptions. To sidestep protections that prevented reading code pages, Blind-ROP [24] employed sophisticated probing techniques to learn pointer values. Specifically, Blind-ROP used a buffer overflow vulnerability to repeat-

	Attacks	Vulnerability/Info. Asset	Source	Acquisition
2013	**Just-in-Time Code Reuse (JIT-ROP) [23]**	Location of Code (I-2)	OS Choices	ENGR*
2014	**Blind ROP [24]**	Location of Code (I-2)	OS Choices	PROBE*
2016	**Crash-Resistant Oriented Programming [94]**	Location of Code (I-2)	OS Choices	ENGR*

Table 2.5: **Novel Vulnerability Sources and Acquisition Methods in Derandomization Attacks.** We enumerate derandomization attacks that have leveraged a novel vulnerability source or method of information acquisition (denoted by *). We classify these assets by their source and method of acquisition. In this genealogy, code location originates from OS Choices because ASLR [5] is in place.

			MAE for CP	Code Contents	Order of Data	Loc. of Data	Order of Code	Loc. of Code	Rep. of Code	Rep. of CP	Rep. of DP
	Attacks										
2004	**Brute Force Attacks on ASLR [89]**		●	◐	◐	◐	○	●	○	●	○
2013	**Just-in-Time Code Reuse (JIT-ROP) [23]**		●	◐	◐	◐	○	●	●	●	○
2014	**Blind ROP [24]**		●	◐	◐	◐	○	●	○	●	○
2016	**Crash-Resistant Oriented Programming [94]**		●	◐	◐	◐	○	●	○	●	○
	Defenses										
2006	**Address Space Layout Permutation [78]**	*Load-Time Obfuscation*	○	○	●	●	●	●	○	○	○
2014	**Oxymoron [90], Isomeron (2015) [91]**	*Load-Time Obfuscation*	○	○	○	○	○	◑	○	●	○
	Execute-no-Read (XnR) [92]	*Runtime Enforcement*	○	○	○	○	○	○	○	●	●
2015	**Readactor [93]**	*Load-Time Obfuscation* *Runtime Enforcement*	○ ○	○ ○	○ ○	○ ○	○ ○	◑ ○	○ ○	● ●	○ ◑
	Timely Addr. Space Randomization (TASR) [57]	*Runtime Obfuscation*	○	○	○	●	○	●	○	○	○
2016	**RuntimeASLR [58]**	*Runtime Obfuscation*	○	○	○	●	○	●	○	○	○
	Remix [59]	*Runtime Obfuscation*	○	○	○	○	◑	◑	○	○	○
	Shuffler [12]	*Runtime Obfuscation*	○	○	○	○	●	●	○	◑	○
2019	**Smokestack [95]**	*Runtime Obfuscation*	○	○	◑	◑	○	○	○	○	○

Table 2.6: **Chronological Ordering of Derandomization Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: memory access error (MAE), code pointer (CP), and data pointer (DP).

edly overwrite a single byte of the return address on the stack, using the execution of the child process as an oracle to determine if the guessed byte was correct. Repeated probing of subsequent bytes disclosed the complete return address, derandomizing the ASLR offset and enabling a ROP attack. Some derandomization techniques leveraged timing side channels

to recover the ASLR offset. We discuss these attacks in Section 2.5.3.4 as they leverage microarchitectural vulnerabilities that are common to the timing side-channel attack genealogy.

2.5.2.2 Defenses with Re-Randomization

In response to derandomization attacks, defenses have employed re-randomization of critical information assets, including re-randomization of the location and order of code/data objects, the representation of code, and the representation of pointers. Although the re-randomization of defenses provably adds one bit of entropy to existing systems [89], recent work has relied heavily on this technique to mitigate derandomization attacks. Re-randomization elevates the difficulty of obtaining implementation information by making it transient, forcing runtime acquisition techniques to learn and exploit information before it changes. New ASLR-variants re-randomized the *location of objects* dynamically, such as after every kernel I/O operation (TASR [57]) or processes fork (RuntimeASLR [58]). Both approaches serve to destroy leaked pointers before they are injected back into the system. Periodic re-randomization has also proven valuable. Morpheus [31] reduces performance overheads by leveraging hardware support to re-randomize memory during runtime, but only obfuscates the absolute location of code and data. Remix [59] obfuscates relative addressing in the code segment by reordering basic blocks within their respective functions, but does not randomize the location of function entries. Conversely, Shuffler [12] obfuscates the relative distance between functions by reordering them at runtime, but does not address intra-function relative distance. Smokestack [95] focuses solely on data, randomizing the stack layout to mitigate memory access errors. Ultimately, runtime re-randomization introduces chaos for attackers attempting to execute code gadgets.

2.5.3 Timing Side-Channel Attacks

Timing side-channel attacks leverage information assets introduced during hardware realization to leak confidential data. These attacks abuse a ***side channel***—a vulnerability originating from the physical processor model where information propagates to externally observable states in an unintended manner. These observable states include shared state (*e.g.,* cache contents) and readable auxiliary state (*e.g.,* program timing). We adopt the following definition to capture this class of attacks:

> ***Timing Side-Channel Attack:*** A class of security exploits that learn some information about a target system or application by observing its timing properties. Timing side

channels can involve passive observation and direct interference with the victim process (*e.g.*, by polluting cache state). Timing side channels are also referred to as timing attacks in the literature and originated as a technique to compromise a cryptosystem by learning secret key information.

Timing side channels can be used as an information acquisition tactic to derandomize systems. Thus, the class of timing side-channel attacks and derandomization attacks intersect. In this work, we categorize side-channel-based derandomization under this genealogy, as discussed in Section 2.5.3.4.

Attackers must have intimate knowledge of code contents to decipher how side channels reveal program secrets during execution. Furthermore, a significant number of timing side-channel attacks directly interfere with the victim program, leveraging *sharing-dependent program timing* to strategically reveal confidential information. We provide a chronological listing of attacks that leverage novel vulnerability sources in Table 2.7, as well as prominent attacks and defenses in Table 2.8. Below, we detail how these attacks have evolved, from early work that exploited program timing, to advanced attacks that leveraged external interference to leak secret information. We also discuss timing side-channel attacks that have sought to derandomize ASLR rather than leak cryptographic keys.

2.5.3.1 Timing Secret-Dependent Operations

In 1996, Kocher observed that the control flow of cryptographic algorithms was key-dependent. Bits of the secret key influenced readable auxiliary states (*e.g.,* program timing) in the implementation, enabling an observer with knowledge of the algorithm to recover these key bits. While the cryptographic primitives were sound, the naive source code implementations proved to be dangerous due to unintended fluctuations in execution time. Kocher demonstrated that secret key information could be obtained by feeding the program carefully designed inputs and measuring the *execution time* of the algorithm [39]. To prevent this exploit, constant-time algorithms with key-independent control flow were developed, effectively *avoiding* side channels in the implementation. In many cases, it was up to the programmer to develop constant-time source code and assure that compilation and hardware realization upheld their intent. This task was incredibly challenging as it required significant expertise to model the complexities of the compiler and hardware in one's notional computer model. Other defenses sought to automate this process [96]. While many constant-time implementations were developed, they were fragile as any small fluctuations in execution could break them.

	Attacks	Vulnerability/Info. Asset	Source	Acquisition
1996	**Kocher's Timing Attacks... [39]**	Side Channels (V-9) Timing, *Execution Time* (I-11)	Phys Proc Model μArch Choices	– PROBE
2005	**Bernstein's Timing Attack [97]**	Microarch. Sharing, *L1-Cache* (I-10) Timing, *Mem. Accesses* (I-11)	μArch Choices μArch Choices	ENGR, PROBE PROBE
	Prime+Probe, Evict+Time [98, 99]	Sharing-Dep. Program Timing (V-7)	Phys Proc Model	–
2007	**BTB Side Channel [100]**	Microarch. Sharing, *BTB* (I-10)	μArch Choices	ENGR, PROBE
	I-Cache Side Channel [101]	Microarch. Sharing, *I-Cache* (I-10)	μArch Choices	ENGR, PROBE
2014	**Flush+Reload [56]**	Microarch. Sharing, *LLC* (I-10)	μArch Choices	ENGR, PROBE

Table 2.7: **Novel Vulnerability Sources in Timing Side-Channel Attacks.** We enumerate timing side-channel attacks that leverage a novel vulnerability or information asset (*i.e.,* one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition. For space driven brevity, we adopt the following abbreviations: microarchitecture (μArch).

Although cryptographic algorithms adopted data-independent control flow, some performed key-dependent memory accesses to critical data structures (*e.g.,* substitution boxes), leaving them vulnerable to attack. Nearly ten years after Kocher's original work, Bernstein published a timing side-channel attack on AES that instead measured timing variations of *memory accesses* using knowledge of *microarchitectural structures* [97]. Bernstein observed capacity misses during key-dependent table lookups to infer which table entries were accessed. By iteratively choosing program input and observing memory composition via timing information, he was able to recover the secret key. This attack specifically observed differences in the timing of memory accesses due to *internal interference* in the victim program. Bernstein's attack inspired future side-channel exploits that observed timing differences caused by *external interference* provoked by the attacker.

2.5.3.2 Timing Memory Accesses

As few algorithms exhibited internal interference, attacks began to instigate timing differences by directly interfering with the victim program. This trend was shaped by widespread support for simultaneous multithreading (SMT) [114], which caused threads to share *all* microarchitecture structures in a processor. Thus, attackers could modify shared state in the implementation FSM by leveraging knowledge of microarchitectural sharing and the location of critical data. Then, attackers could use *sharing-dependent program timing* to observe the result of this external interference in one of two ways: via timing-based or access-based attacks.

In timing-based attacks, attackers modify state shared between the victim's and attacker's

			Side Channels	SD Prog. Timing	Code Contents	Timing of Op.	Loc. of Data	Loc. of Code	Microarch. Sharing
	Attacks								
1996	**Kocher's Timing Attacks... [39]**		●	○	◐	●	○	○	○
2005	**Bernstein's Timing Attack on AES [97]**		●	○	◐	●	○	○	◐
	Cache Missing, Prime+Probe, Evict+Time [98, 99]		●	●	◐	●	◐	○	◐
2007	**BTB Side Channel [100]**		●	●	◐	●	○	◐	◐
	I-Cache Side Channel [101]		●	●	◐	●	○	◐	◐
2013	**Attacks Against Kernel Space ASLR [102]**		●	●	◐	●	○	◐	◐
2014	**Flush+Reload [56]**		●	●	◐	●	◐	○	◐
2016	**Jump over ASLR [26]**		●	●	◐	●	○	◐	◐
2017	**ASLR⊕Cache (AnC) [25]**		●	●	◐	●	○	◐	◐
2021	**Frontal [103]**		●	○	◐	●	○	◐	◐
	Defenses								
2005	**Automated Constant-Time Programs [96]**	*Program-Time Avoidance*	◐	○	○	○	○	○	○
2007	**Random Permutation Cache [104]**	*Runtime Obfuscation*	○	○	○	○	○	○	◐
	Partition Locked Cache [104]	*Runtime Enforcement*	○	◐	○	○	○	○	◐
2013	**Path ORAM [105]**	*Program-Time Avoidance*	◐	○	○	○	○	○	○
2016	**Cache Allocation Technology (CAT) [106]**	*Runtime Enforcement*	○	◐	○	○	○	○	◐
2018	**Dynamically Allocated Way Guard (DAWG) [107]**	*Runtime Enforcement*	○	◐	○	○	○	○	●
	CEASER [108]	*Runtime Obfuscation*	○	○	○	○	○	○	◐
2019	**Data-Oblivious ISA Extensions (OISA) [109]**	*Runtime Enforcement*	●	○	○	○	○	○	○
	Skewed-CEASER (CEASER-S) [110]	*Runtime Obfuscation*	○	○	○	○	○	○	◐
2020	**PhantomCache [111]**	*Runtime Obfuscation*	○	○	○	○	○	○	◐
2021	**Mirage [112]**	*Runtime Obfuscation*	○	○	○	○	○	○	◐
2022	**DAGguise [113]**	*Runtime Obfuscation*	○	○	○	○	○	○	◐

Table 2.8: **Chronological Ordering of Timing Side-Channel Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent side-channel attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviation: sharing-dependent (SD).

implementation FSM, then time the victim's execution to see the effects of their malicious interference. These attacks specifically exploit readable auxiliary state in the victim's implementation FSM to leak secrets. In access-based attacks, attackers similarly influence shared state, then allow the victim program to execute. However, the attacker does not observe

the victim's execution via auxiliary state. Rather, the attacker executes and times their own program (*i.e.,* the attacker's implementation FSM) to see how the victim has modified shared state. Because the attacker measures their own memory accesses, they can obtain timing information at a much finer granularity and higher fidelity than observing the victim.

In 2005, Percival and Osvik et al. independently formalized access-based attacks on RSA and AES that leveraged sharing in the L1 cache [98, 99]. These exploits first primed the cache by filling it with attacker data, essentially providing attackers with complete knowledge of the shared cache state. Afterwards, attackers surrendered execution to the victim process. The victim process loads confidential data into the cache, modifying the shared state in the attacker's implementation FSM. Once the attacker resumes control of the processor, they can recover these traces by timing their own memory accesses to infer shared state. Slow accesses signified a conflict miss, revealing that the victim program accessed data mapped to the same cache block. By compiling these access patterns, the secret key was successfully recovered. Osvik published a second attack, termed Evict+Time, that took a timing-based approach to break AES. Rather than probing memory and measuring the time of each access, the victim's overall execution time was measured after the cache was evicted. Slowed execution signified that the cryptographic algorithm used the evicted data, revealing the memory access patterns of the algorithm.

2.5.3.3 Beyond the L1 Cache

Attackers continued to uncover alternative side channels to observe a victim program's execution. Given implementation information about *microarchitectural sharing* in the Branch Target Buffer (BTB), Aciicmez et. al observed the branching patterns of a cryptographic process [100]. Aciicmez also extended this technique to the I-cache to similarly monitor the control flow of the victim process [101]. In both cases, the attacker was able to derive the outcome of a key-dependent branch condition, leaking information about the secret key. Timing side-channel exploits also expanded into lower levels of the memory hierarchy. A new cache attack called Flush+Reload [56] achieved high resolution in the last-level cache (LLC) by exploiting shared memory between two cores. This technique can be used to break process isolation and trace memory accesses of a victim. More importantly, this exploit broadened the attack surface from processes running on the same core to processes running on separate cores. Thus, disabling SMT could not stop this attack. Subsequent work demonstrated that Prime+Probe is also practical on the LLC [53]. Recent attacks have moved to exploit timing subtitles in the CPU frontend and have been able to use this technique to leak secrets from Intel SGX enclaves [103].

2.5.3.4 Side-Channel-Based Derandomization

Timing side-channel attacks have also been used to derandomize ASLR [5]. In 2013, the I-cache side-channel attack [101] was used to dismantle kernel-space ASLR [102]. Collisions in the I-cache revealed the address of the system call handler and, through repetition, this information was combined to uncover bits of the KASLR offset. Jump-over-ASLR exploited a timing-side channel in the BTB to monitor collisions between the attacker's branch instructions and the victim program, eventually derandomizing the lower bits of ASLR [26]. The following year, AnC used Evict+Time [99] to observe the *page table walk* of the memory management unit. Ultimately, this attack was able to derive 28 bits of the ASLR offset in only 150 seconds [25]. This attack combined traditional side channels with a unique probing technique to recover more bits of ASLR than previous exploits in a short amount of time.

2.5.3.5 Defenses for Timing Side Channels

As shown in Table 2.8, defenses for timing side-channel attacks have broadly taken one of the following approaches: *i)* Randomization of microarchitectural sharing, *ii)* Enforcement of microarchitectural sharing/sharing-dependent behaviors via the partitioning of shared structures, *iii)* Randomization of program timing via adding noise to timing infrastructures, or *iv)* Avoidance/enforcement of side channels via the use of constant-time or data-oblivious programs. Novel cache architectures obfuscate microarchitectural sharing by randomizing memory mappings within shared structures [104]. CEASER [108] and Skewed-CEASER [110] recently further expanded obfuscation of sharing by using encrypted addresses for cache indexing. These two techniques made it improbable to craft an eviction set by dynamically remapping the caches with high entropy and were applicable for the LLC.

Other protections use enforcement to protect microarchitectural sharing and sharing-dependent program timing by prohibiting external interference in the cache. These mitigations include PL-Cache [104], which proposed secure cache replacement protocols that prevented confidential data from being evicted by other processes. Subsequent protections continued to address inter-process conflicts. Cache Allocation Technology (CAT) [106] dynamically partitioned the LLC, enforcing strict bounds between two processes to mitigate Flush+Reload attacks. This defense was extended to all set associative structures, including caches, in Dynamically Allocated Way Guard (DAWG) [107].

Recent work has revisited mitigating side channels via program avoidance through data-oblivious programming. Data-oblivious programming is broader than constant-time programming, as it not only considers a program's execution time but also considers its memory access patterns and control flow. Data-Oblivious ISA Extensions (OISA) [109] enforces data-

oblivious guarantees in the instruction-set architecture directly. Specifically, OISA detects and prevents confidential data from propagating to side channels at runtime, effectively enforcing constant-time code. Oblivious RAMs (ORAMs) implement data-independent memory lookups to avoid timing side channels based on memory access patterns [105, 115, 116]. By integrating these algorithms, applications can perform secret-dependent memory accesses without revealing secrets via side channels. Recent work has obfuscated memory lookups in DRAM controllers using similar strategies [113]. In Chapters 7-9, we discuss data-oblivious programming strategies in more detail, as this technique is required by some of our proposed encrypted architectures.

2.5.4 Transient Execution Attacks

A new and dominant class of attacks exfiltrate secrets by observing the side effects of transient instructions that operate on sensitive information. These attacks abuse *mispeculation*—the spurious execution of an instruction that is eventually deemed unnecessary due to the emergence of a misprediction or fault. When mispeculation occurs, modern processors squash all in-flight instructions and revert the processor state. However, not *all* state is reverted. Residues in the caches and other shared microarchitectural structures remain, as these states do not affect program correctness. This oversight establishes a *side channel* for third parties with adequate microarchitectural knowledge to observe program execution. We adopt the following definition to capture this genealogy of attacks:

> ***Transient Execution Attack:*** A class of security exploits that abuse speculative execution to leak secret data to observable outputs. Various techniques can be used to recover these secrets, including timing side channels that have already been discussed (*e.g.*, FLUSH+RELOAD [56] in Section 2.5.3.3). This genealogy is characterized by its abuse of speculative execution, not by its secret recovery techniques. The term "transient instructions" was coined by Spectre [16] in 2018.

Transient execution attacks are broadly categorized into two classes: those that cause mispredictions by polluting the contents of the branch predictor, and those that leverage faults to indirectly squash instructions [117]. We provide a chronological listing of attacks that have leveraged novel vulnerability sources in Table 2.9 and prominent transient execution attacks and defenses in Table 2.10. Below, we detail how this prominent new class of attacks exploits microarchitectural design choices in sophisticated ways.

	Attacks	Vulnerability/Info. Asset	Source	Acquisition
2018	**Spectre [16]**	Mispeculation, *Prediction-Based* (V-8) Sharing-Dep. Program Exec., *BTB* (V-6)	Phys Proc Model Phys Proc Model	– –
	Meltdown [17]	Mispeculation, *Fault-Based* (V-8)	Phys Proc Model	–
	Foreshadow [118]	Microarch. Sharing, *SGX Cache* (I-10)	μArch Choices	ENGR, PROBE
2019	**RIDL [119], Fallout [120]**	Microarch. Sharing, *Store Buffer* (I-10)	μArch Choices	ENGR, PROBE

Table 2.9: **Novel Vulnerability Sources in Transient Execution Attacks.** We enumerate transient execution attacks that have leveraged a novel vulnerability or information assets (*i.e.,* one that has not been used previously). We classify these vulnerability sources by their source and method of acquisition. For space driven brevity, we adopt the following abbreviations: microarchitecture (μArch).

	Attacks		Mispeculation	SD Prog. Exec.	Side Channels	SD Prog. Timing	Code Contents	Timing of Op.	Loc. of Data	Loc. of Code	Microarch. Sharing
2018	**Spectre V1 [16]**		●	●	●	●	◐	●	◐	◐	◐
	Meltdown [17]		●	○	●	●	◐	●	◐	○	◐
	Foreshadow [118]		●	○	●	●	◐	●	◐	○	◐
2019	**RIDL [119], Fallout [120]**		●	○	●	●	◐	●	◐	○	◐
2020	**LVI Control Hijack [4]**		●	●	○	○	◐	○	○	◐	◐
	Defenses										
2018	**Retpoline [121, 122]**	*Program-Time Enforcement*	◐	○	○	○	○	○	○	○	○
	InvisiSpec [123]	*Runtime Enforcement*	◐	○	◐	○	○	○	○	○	○
2019	**Speculative Taint Tracking [124], NDA [125]**	*Runtime Enforcement*	●	○	◐	○	○	○	○	○	○
	CleanupSpec [124]	*Program-Time Enforcement*	◐	○	◐	○	○	○	○	○	○
	MI6 [126]	*Runtime Enforcement*	◐	○	○	○	○	○	○	○	●

Table 2.10: **Chronological Ordering of Transient Execution Attacks and Defenses.** We enumerate the vulnerabilities and information assets leveraged/protected by prominent transient execution attacks and defenses. Each asset is either leveraged/protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviation: sharing-dependent (SD).

2.5.4.1 Prediction-Based Attacks

Spectre [16] directly influences the misprediction of instructions by exploiting *sharing-dependent program execution* in the branch target buffer (BTB). Leveraging knowledge of *microarchitectural sharing*, the attacker trains the BTB to poison the shared predictor and

force the misprediction of a specific branch in the victim's program. In Spectre V1, the victim mispredicts a conditional branch that makes a secret-dependent memory access, leaving residues in the cache. The evidence of this transient execution is later recovered by the attacker via either PRIME+PROBE [98] or FLUSH+RELOAD [56]. We show the vulnerabilities and information assets leveraged by Spectre V1 in Table 2.10, which is comprised of the aforementioned assets, as well as the location of the spectre gadget and the assets used by PRIME+PROBE/FLUSH+RELOAD. Spectre V2 forces a misprediction on an indirect branch, enabling the attacker to redirect program execution to *any* unintended code of their choosing. This exploitation mimics ROP [55] and would require a similar set of attack assets.

Mispeculation provides attackers an avenue to directly influence the execution of a victim program without requiring program-level vulnerabilities. However, because mispeculated instructions are eventually squashed, this hardware-level vulnerability can only be used to execute short instruction sequences. Spectre-style exploits have since expanded to other microarchitectural structures, including the pattern history table, return stack buffer, and memory disambiguation logic [127, 128, 129].

2.5.4.2 Fault-Based Attacks

Rather than influencing the misprediction of an instruction, Meltdown-style attacks rely on faulty operations that squash transient instructions [117]. Meltdown abuses the fact that memory permissions in specific Intel processors are not checked until instruction commitment [17]. Thus, memory accesses and other invalid operations are performed long before they are deemed faulty. Meltdown leverages this property to leak kernel memory by performing transient kernel reads, which are later recovered through cache timing side channels like FLUSH+RELOAD [56]. Because Meltdown does not influence program execution via shared state, it does not leverage sharing-dependent program execution like Spectre [16].

Foreshadow [118] similarly exploited delayed permissions checks to subvert Intel SGX [130]. Specifically, Foreshadow makes a transient access to enclave memory during an address translation with a faulty page-table lookup. By using timing side channels to observe the cache state after the fault, attackers were able to derive SGX enclave secrets. Fault-based attacks expanded to include variants that raise exceptions due to accesses to system registers or lazy floating-point register state switching [131], and have extended into the store buffer [119, 120].

LVI [4] exploits mispeculation to *inject* malicious values into the pipeline via forwarding in microarchitectural structures. During transient execution, the victim program computes on this "poisoned" data before rollback. In Table 2.10, we list the LVI variant that redirects control flow to a malicious code gadget by replacing a return address with a malicious

address. This variant exploits code contents and locations, similar to control-flow attacks. Other variants of the LVI attack use traditional timing side channels to recover residues of the transient execution, requiring similar assets to Spectre [16].

2.5.4.3 Defenses for Transient Execution Attacks

To combat transient execution attacks, defenses have focused on addressing the hardware-level constructs that permit the exploit. Specifically, defenses have broadly taken two approaches: *i)* Limiting the speculative execution of instructions, classified as either avoidance or enforcement of mispeculation, or *ii)* Eliminating the side-effects of transient instructions, classified as the enforcement of mispeculation and side channels.

Retpoline [121, 122] uses software enforcement techniques to prevent the speculative execution of vulnerable instructions. Specifically, Retpoline uses a return trampoline to prevent the processor from speculating on an indirect jump, thus prohibiting attackers from steering code during speculation. Other mitigations address the microarchitectural side-effects left by transient execution, preventing this unintended information from propagating to shared structures. We classify this technique as the enforcement of mispeculation and side channels, as it prevents sensitive information from propagating to unintended outlets (*i.e.*, side channel vulnerability) during mispeculation. InvisiSpec [123] prevents transient loads from propagating to the cache state, whereas similar defenses monitor and restrict the propagation of speculative data from reaching any known covert channels [125, 132]. Other protections try to address covert channels through enforcement and partitioning [107, 133, 134]. Rather than preventing the propagation of sensitive information, CleanupSpec [124] *undoes* microarchitectural residues that occur during mispeculation, essentially reverting to the same state in the implementation FSM before mispeculation. Thus, even though transient execution occurs, the processor truly reverts to the previous state, preventing attackers from exfiltrating secrets.

2.6 Discussion and Limitations

This chapter aims to address the significant challenge of understanding the large space of security exploits and defenses by presenting an organization of security vulnerabilities as they arise throughout the program life cycle. This framework is designed with the purpose of conceptual modeling to aid researchers in developing more durable defenses that protect against wider classes of attacks. Specifically, this framework provides researchers a way to organize vulnerability sources across phases and sources within the program life cycle, identify similarities across different exploits, and model the ways in which defenses address

these vulnerability sources. We do not intend for this framework to be used to model security attacks and defenses using design exploration tools. In fact, given the complexity of the state space of our proposed FSM framework, such modeling may very well be infeasible and thus is not the intent of this work.

As presented in this chapter, the program life cycle framework captures security exploits that compromise program confidentiality or integrity. Availability exploits, such as denial-of-service attacks, are beyond the scope of our current framework. However, some denial-of-service attacks are easy to consider in this framework. For example, a denial-of-service attack that uses a malicious packet to crash a program is clearly transitioning the program into an unintended state, in alignment with our proposed framework. Other availability exploits are more complex and require additional consideration, such as those that flood a server with packets to oversaturate server capacity and hinder performance. While the results of such an attack are also unintended by the programmer, they require more careful consideration of the programmer's intended FSM to fully model how these exploits violate unintended states and transitions.

Finally, our framework uses auxiliary state (Section 2.2.3.3) to capture time and occurrence count, enabling us to model analog effects. However, auxiliary state is hard to discover in advance of deploying a system, as these effects are not intentional. Rather, these states are often determined *post-facto*, only after their value has been demonstrated. Thus, it is challenging to identify vulnerability sources that exploit auxiliary state within our framework.

2.6.1 Defense Design Trends

Mitigating attacks through the *enforcement* of vulnerabilities remains a popular approach. Certainly, many solutions exist to enforce memory safety through the detection of memory access errors [60, 76]. Additionally, the enforcement of memory permissions has seen steady advancements, from the NX-bit stack protections [70], to x86 SMAP and SMEP [135], and more recently Intel SGX [130]. Remaining program-level vulnerabilities (*e.g.,* uninitialized values and integer overflow) can be detected using generalized debugging tools, like as UBSan [136] and ASan [137]. However, unless these enforcement mechanisms are supported by specialized hardware, performance overheads become high, leaving most code running in the wild with few protections from these vulnerabilities.

For *obfuscation* defenses, two trends have emerged: *i)* The movement toward the randomization of multiple pieces of information assets, and *ii)* The application of finer-grained randomization defenses. Initially, randomization was reserved for the location of objects

(*e.g.,* ASLR [5]). But, over time, randomization defenses have expanded to include the representation of code [72], representation of pointers [28, 31], microarchitectural sharing [108], timing of operations [138], relative location of functions [12], and the order of stack variables [95]. These techniques garner broad appeal, since they (at least partially) address a wide range of attacks that utilize the randomized information assets. Furthermore, defenses have also trended toward more fine-grained, runtime randomization of information assets. In its most course-grained form, randomization is applied at compile-time or load-time. Thus, long periods are available to probe these randomized systems, giving attackers a foothold to discover the obfuscated information assets. Derandomization attacks of this nature have been extremely successful (*e.g.,* Blind ROP [24] and AnC [25]). Consequently, defenses have begun to aggressively re-randomize during execution in hopes that this approach will defeat malicious attempts to acquire information assets at runtime.

2.6.2 Performance-Security Trade-Offs

While defenses may appear promising or protect against a wide array of vulnerability sources, the costs of these defenses must also be considered. In this context, cost may refer to runtime performance overheads, complexity of software modifications, complexity of hardware modifications, silicon area and power overheads, cost to the programmer to use the defense, *etc.* Existing security solutions that are deployed in practice are those that presented the best cost trade-offs at that time, otherwise they would have been dismissed. For example, ASLR [5] was one of the first widely-deployed control-flow defenses and had very low cost. However, ASLR has since proven to be non-durable, as many exploits have prevailed despite it by leveraging advanced derandomization attacks (Section 2.5.2). Many ASLR variants have since been proposed but none have been widely deployed.

Some security solutions are known but deemed impractical due to their high costs, including some discussed in this chapter (*e.g.*, eliminating speculative execution or completely isolating processes for timing side channels). It is possible that these high-cost defenses will never be deployed or will only be deployed when a pervasive attack makes going without them intolerable. It is often the case that security can be sacrificed to lessen costs, or that certain costs can alleviate others. For example, modified computer hardware is often leveraged to reduce the performance overheads of runtime defenses [31]. In this case, hardware complexity and silicon area and power overheads are increased to lessen performance overheads. The framework proposed in this chapter serves to aid the development of efficient, comprehensive defenses by modeling similarities in vulnerability sources across prominent security attacks. To assess the cost trade-offs of new defenses, it is also critical to understand

what threat model is necessary and what costs are tolerable to developers and consumers. Both are incredibly challenging tasks that warrant future work.

2.6.3 Suggestions for Future Defenses

Our study reveals two key properties that can inform how to best approach secure system design in the future: *i)* A small amount of implementation information is used widely across numerous security attacks, and *ii)* The classes of vulnerabilities grow very slowly, with only a few new vulnerability classes introduced per decade. Given these observations, defenses that focus on enforcing or obfuscating *multiple* information assets have the potential to be both effective and durable, as broad swaths of the attack landscape will be disrupted. In contrast, avoidance defenses that focus on patching vulnerabilities will become increasingly ineffective and non-durable. Their ineffectiveness arises from the challenge of finding all vulnerabilities, which requires formal methods that do not scale with large systems. Additionally, the introduction of new code can introduce a new vulnerability, and no strategies exist for finding yet-undiscovered vulnerabilities.

Enforcement and obfuscation are appropriate techniques are both effective at providing security. Enforcement defenses can be made sound and complete, unlike obfuscation defenses, which provide only probabilistic security guarantees. Yet, enforcement defenses may conflict with non-malicious programs that use undefined semantics [60], while obfuscation techniques can often comply with these requirements [31]. Perhaps the defense approach that emerges predominantly will depend on the efficiency of the defense and to what extent they impact programmers that utilize these machines.

For vulnerabilities introduced during hardware realization, protections have focused largely on randomization-based techniques. We see randomization defenses trending toward higher-entropy implementations to increase their overall strength against brute-force and side-channel attacks. As detailed in Section 2.5.2, derandomization attacks have progressed to the point where randomized semantics can be recovered with mere minutes of probing. Higher entropy protections will diminish the likelihood of guessing randomized information, and while derandomization attacks will still be possible, they will take much more time. Thus, combining high-entropy defenses with fast rates of re-randomization will undercut the expected time of derandomization attacks and ensure that these systems can never be successfully penetrated through accessing implementation information. Alternatively, defenses could leverage *encryption*, which is more resilient to cryptanalysis and other disclosure mechanisms. With cryptographic protection of assets, attackers are forced to probe systems to acquire information. Such probing can be thwarted via re-randomization

and side-channel-resilient design.

Finally, our framework highlights the importance of modeling auxiliary states, such as observable timing or occurrence counters, to capture analog effects that affect visible state, like charge loss. Thus far, such auxiliary states have been determined *post-facto*, only after their value has been demonstrated. An important defense strategy would be to *anticipate* auxiliary states that could be exploited in the future—a very hard problem—but one that must be tackled in this ever-evolving arms race between attacks and defenses.

2.7 Chapter Summary

In this chapter, we examined how vulnerability sources are introduced throughout the program life cycle, and how these sources, along with information assets, are used by attackers to construct security exploits. These vulnerability sources arise from programming, compilation, and hardware realization, and introduce states in the implementation FSM that are *unintended* relative to the programmer's intended FSM. Security exploits leverage these sources to transition the implementation FSM to an unintended state to carry out an attack. Further, through the case studies of four important genealogies, we saw the evolution and increasing sophistication of the implementation information used for attacks and the methods to acquire this information, alongside a corresponding increase in the sophistication of defenses. We conclude by drawing on our framework and genealogies to point to key trends in defenses that hold increasing promise for the future.

	Defenses	Chapter		Order of Data	Loc. of Data	Order of Code	Loc. of Code	Rep. of Code	Rep. of CP	Rep. of DP	Rep. of Data
2019	**Morpheus [31]**	***Ch. 3***	*Runtime Obfuscation*	○	●	○	●	○	○	○	○
			Runtime Encryption	○	○	○	○	●	●	●	○
2020	**Displaced & Dilated Address Space [32]**	***Ch. 4***	*Runtime Obfuscation*	○	○	●	●	○	○	○	○
2021	**Morpheus II [33]**	***Ch. 5***	*Runtime Encryption*	○	○	○	○	●	●	○	○
2022	**Sequestered Encryption [35]**	***Ch. 7, Ch. 8***	*Runtime Encryption*	○	○	○	○	○	○	○	●

Table 2.11: **Chronological Ordering of Defenses Presented in this Dissertation.** We enumerate the information assets protected by the defenses presented in this dissertation. Each asset is either protected fully (●), partially (◐), or not at all (○). For space driven brevity, we adopt the following abbreviations: code pointer (CP), and data pointer (DP).

2.8 Implications for This Dissertation

The remainder of this dissertation explores ways in which encrypted architectures can provide durable security and privacy protections by guarding information assets. This approach is motivated by the insights in this chapter and evolves as we gain experience and understanding of building information asset protections. Our initial work continues the trend observed in Section 2.6: hardware-based protections typically obfuscate assets and are trending toward higher-entropy defenses. Our first defense utilizes this approach almost directly. In subsequent chapters, we embrace cryptography to better thwart information acquisition techniques and ultimately trade re-randomization for side-channel-resilient design to holistically thwart attempts to learn protected assets. Throughout this evolution, we continually leverage custom hardware to produce defenses that are comprehensive while reducing runtime performance overheads. Table 2.11 summarizes the defenses presented in this dissertation, organizing them under the format used in our case studies.

CHAPTER 3

Architectural Approaches to Protecting Code and Pointers

Attacks succeed by abusing a vulnerability and system implementation information to force the victim program into an unintended state. Our secure hardware architecture, Morpheus, combines two powerful protection mechanisms—ensembles of moving target defenses and churn—to protect information assets, thereby inhibiting attacks. Ensembles of moving target defenses randomize key program values (*e.g.*, relocating pointers and encrypting code and pointers) which forces attackers to extensively probe the system prior to an attack. To ensure attack probes fail, the architecture incorporates churn to transparently re-randomize program values underneath the running system. With frequent churn, systems quickly become impractically difficult to penetrate. We demonstrate Morpheus through a RISC-V-based prototype designed to stop control-flow attacks. Each moving target defense in Morpheus uses hardware support to individually offer more randomness at a lower cost than previous techniques. When combined with churn, Morpheus defenses offer strong protection against control-flow attacks, with our security testing and performance studies revealing: *i)* High-coverage protection for a broad array of control-flow attacks, including protections for advanced attacks and an attack disclosed after the design of Morpheus, and *ii)* Negligible performance impacts (1%) with churn periods up to 50 ms, which our study estimates to be at least 5000× faster than the time necessary to possibly penetrate Morpheus.

The research presented in this chapter appears in:

[31] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, Todd Austin. ***"Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn,"*** Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019. DOI: 10.1145/3297858.3304037.

3.1 Introduction

Building vulnerability-free, complex systems is challenging and further complicated by legacy code and changing threat models. Control-flow exploits are still prevalent [139] despite several layers of defenses based on safe pointers [140], control- and data-flow integrity [61, 76, 141], limiting the use of data as code (NX-bit) [51], and address space layout randomization (ASLR) [5]. A similar arms race is being played out in the microarchitecture, where side channels are abusing speculation [16] and memory characteristics [98] to circumvent architectural isolation guarantees.

Rather than trying to insulate complex systems from vulnerabilities, we assume that the system contains vulnerabilities and attackers are attempting to penetrate via those vulnerabilities. Instead, we propose protecting the information assets that are critical to craft successful attacks, using hardware to enable an *ensemble* of such moving target defenses to be deployed concurrently. Additionally, we introduce *churn*, a mechanism that transparently re-randomizes the moving target program values while the system is in operation. By limiting our defenses to values associated with undefined program semantics, we construct a secure system that is transparent to normal programs but intentionally hostile to malicious programs. Crucially, even though concrete exploits and vulnerabilities grow rapidly, *information assets do not*. Thus, this design approach presents a more durable and proactive layer of defense than patching specific vulnerabilities.

We designed the ***Morpheus Secure Architecture*** [31, 33] guided by these insights. Morpheus is a RISC-V-based system that incorporates ensembles of moving target defenses (EMTDs) with churn to thwart control-flow attacks on the control and data plane. Morpheus protects critical information by using two novel moving target defenses: ***address space displacement***, which randomly and independently places code and data in the address space, and ***domain encryption***, which randomizes the representation of code, code pointers, and data pointers using strong encryption. These new defenses provide a significantly larger randomization space at lower overheads than previous techniques. Specifically, Morpheus defenses have a randomization space of 504 bits (referred to as "bits of entropy" for the remainder of this work), leading to 2^{504} possible key configurations.

Once deployed, EMTDs force attackers to extensively probe the system to discover the randomized values needed for an attack. Morpheus sabotages the attacker's ability to probe the system and gain critical values by using churn to re-randomize these values at runtime. To further strengthen the value of churn, Morpheus incorporates an ***attack detector*** to sense when an attack is ongoing and immediately trigger a churn cycle to halt the attack. The Morpheus concept is summarized in Figure 3.2.

```
1  void target() {
2     printf("You overflowed successfully, gg");
3     exit(0);
4  }
5
6  void vulnerable(char* str1) {
7     char buf[5];
8     strcpy(buf, str1);
9  }
10
11 int main() {
12    vulnerable("ffffffffffffffff\xf0\x03\x02\x01");
13    printf("This only prints in normal control flow");
14 }
```

Figure 3.1: **Example Control-Flow Attack.** This code exploits a buffer overflow vulnerability in `vulnerable()` to overwrite a return address. We note that the attack code utilizes undefined semantics associated with array overflows, stack frame organization, and code addresses. Morpheus randomizes values associated with undefined semantics to thwart malicious attacks while remaining benign to normal programs.

How Morpheus Stops Attacks. To illustrate our approach, consider the code reuse attack shown in Figure 3.1. This program exploits a buffer overflow vulnerability on Line 8 (V-1) to redirect execution to the function `target()`. The attack payload is shown on Line 12, the synthesis of which requires knowledge of the stack layout (I-5) and the address of the target function (I-2, I-6). When running atop the Morpheus architecture, it becomes intractable to express the attack payload for this program. Under Morpheus, the absolute location of the target (I-2) has been randomized under address space displacement. Further, the representation of code pointers (I-6) is encrypted. The attack payload shown is no longer valid, as the value `0x010203f0` will be "decrypted" and displaced before use, producing a garbled result. To express a valid pointer to `target()`, the attacker must either brute-force guess the information or probe Morpheus to learn the randomized assets. As shown in Section 3.6, derandomizing the Morpheus protections take several minutes or more with the most advanced probes. Unfortunately for the attacker, churn re-randomizes the code locations and pointer representations every 50 ms, producing a new system that the attacker has never seen before and will never seen again. Thus, Morpheus' randomization of information assets with churn thwarts control-flow attacks by preventing adversaries from acquiring the information assets needed to synthesize them.

3.1.1 Chapter Contributions

We believe that architectures can provide security by obfuscating the implementation information needed by attacks. In support of this argument, this chapter makes the following contributions:

- We present Morpheus—a RISC-V-based secure architecture that layers code and pointer protections in an ensemble of moving target defenses (EMTDs) to thwart control-flow attacks. Using compiler, architecture, and runtime support, we implement EMTDs with support for attack-detector-driven churn periods. Our randomization of code, code pointers, and data pointers create significant barriers to control-flow attacks.
- We show that Morpheus, with 504 bits of entropy and 50 ms churn period, has a 1% average slowdown (7% worst case) for SPEC CPU2006 and MiBench benchmarks, despite re-randomizing key program values 5000x faster than estimated attack times.
- We show that Morpheus provides strong security against control-flow attacks, stopping a broad array of advanced attacks including return-oriented programming [55] and the Back-Call-Site attack [62], which was disclosed after the design of Morpheus. These analyses demonstrate that the Morpheus architecture is an effective defense against known and potentially future control-flow attacks.

This chapter is organized as follows. Section 3.2 discusses Morpheus' general approach to protecting code and pointers via EMTDs. In Section 3.3, we detail the implementation of Morpheus on a RISC-V processor. Section 3.4 discusses our prototype of Morpheus with Sections 3.5 and 3.6 presenting our performance and security analyses, respectively. Section 3.7 discusses the limitations of our approach. Finally, Sections 3.8 and 3.9 conclude this chapter.

3.2 Protecting Information Assets with Morpheus

Morpheus is a vulnerability-agnostic, secure architecture that leverages an *ensemble* of moving target defenses (EMTDs) to protect three implementation information domains common across control-flow attacks—***code***, ***code pointers***, and ***data pointers***. To protect these domains, Morpheus layers two novel moving target defenses: ***address space displacement***, which obscures addresses by adding a random displacement to all pointers in the address space, and ***domain encryption***, which encrypts code, code pointers, and data pointers under individual domain keys. This ensemble of defenses used in Morpheus is illustrated in

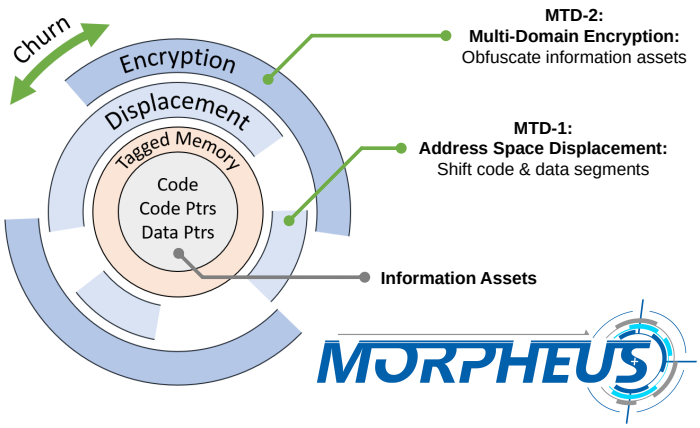


Figure 3.2: **Morpheus' Moving Target Defenses (MTDs).** At its core, Morpheus protects multiple *assets* from attackers, which are identified at runtime using *tagged memory*. The protections are layered on top of these assets: *displacement* and *encryption*. Finally, *churn* drives these two defenses to re-randomize them live.

Figure 3.2. Together, these defenses provide a significantly larger randomization space at lower overheads than previous techniques. Specifically, Morpheus defenses 504 bits entropy, leading to 2^{504} possible key configurations.

We further elevate moving target defenses by combining them with a runtime re-randomization technology called ***churn***. During program execution, churn periodically re-keys the moving target defenses, changing the pointer displacement and re-encrypting protected assets with newly generated secret keys. After this re-keying, all instructions and pointers in the system take on a new form, ensuring that any valuable information gathered by the attackers since the last churn cycle will be lost. This approach combats information leakage and creates high-entropy situations where attackers are confronted with a system they have never seen before and will never see again.

3.2.1 Security Goals

Our goal is to obstruct the acquisition of code and pointer implementation information to prevent the synthesis of control-flow attacks. Specifically, we aim to prevent the attacker from learning enough information about the victim's code and pointers to create a specific sequence of software inputs that, when provided to the victim application, without any assistance from hardware modifications (*e.g.*, Trojans), triggers a vulnerability to redirect program control flow. Specifically, we protect the following domains of implementation information:

- ***Code:*** An instruction, including the opcode (*e.g.*, `JALR`) and operands (*e.g.*, `a0`).

- ***Code Pointers:*** The stored address of a code object, including function pointers, return addresses, and any addresses in memory (*e.g.*, jump tables)
- ***Data Pointers:*** The stored address of a data object, including array pointers and stack pointers

To achieve this goal, we randomize absolute memory addresses via displacement (I-2, I-4) and the representation of code, code pointers, and data pointers via encryption (I-6, I-7, I-8). We do not aim to prevent the creation of attack payloads that do not require knowledge of code or pointer information.

3.2.2 Threat Model

Morpheus is designed to mitigate control-flow attacks, many of which utilize memory exploits [142]. Specifically, we seek to thwart attacks where a trusted but vulnerable victim program processes untrusted inputs that exploit memory access errors (Table 2.1) to hijack control flow of the victim. These attacks may make use of buffer overflows, format-string, heap-spray, double-free, return/jump-oriented programming, and other execution-level semantics to override language-level protections and hijack control flow. We assume a software-based adversary that can interact with the system via an interface such as the network or keyboard. The attacker cannot modify the original binary but can analyze it to locate a memory corruption or disclosure vulnerability in the target application. The attacker has complete knowledge of the system's software, architecture, and microarchitecture, including the Morpheus protections, but does not have knowledge of secret key information (*i.e.*, the domain encryption keys and pointer displacement values). The trusted computing base (TCB) includes the specialized Morpheus hardware, as well as some software support from the loader and OS scheduler.

Non-Goals. We assume that the adversary does not have physical access to the machine and therefore cannot leverage physical attacks, physical side channels (*e.g.*, power), or transistor-scale voltage measurement (*e.g.*, focused ion beam sensing). We also assume that the attacker cannot manipulate the system's boot sequence or anticipate the output of the random number generator. Denial-of-service (DoS) and side-channel attacks are outside the scope of this work.

3.2.3 Displacing Program Segments

To boost uncertainty in memory locations, Morpheus leverages address space displacement to randomize the absolute location of code and data objects at runtime. Morpheus creates two superimposed address spaces above the virtual address space (VAS), one for code (DAS_C) and one for data (DAS_D), and randomly displaces the code and data segments into these spaces. The code segment is displaced by d_{CP} bytes in DAS_C, and the data segment is displaced by d_{DP} bytes in DAS_D, as shown in Figure 3.5. Accordingly, all code/data pointers are incremented by d_{CP}/d_{CP} at runtime. Only this pointer representation is changed; the processor's caches and microarchitectural structures are still indexed by the VAS address. Thus, address space displacement does not affect the location of code and data objects in memory like ASLR [5]. When DAS pointers are used, such as to perform a memory access, this displacement is subtracted off to reveal the virtual address. For a memory request with the location A_{das_D} is transformed through the following computation:

DAS→VAS Translation for Data Pointers:

$$A_{vas} = A_{das_D} - d_{DP} \tag{3.1}$$

Similarly, when a pointer is created, such as during a `JALR` instruction, the value is incremented by the appropriate displacement:

VAS→DAS Translation for Code Pointers:

$$A_{das_C} = A_{vas} + d_{CP} \tag{3.2}$$

Pointer arithmetic (*e.g.*, incrementing a data pointer by an offset) can be performed in the DAS and produce the correct VAS pointer after conversion. Displacement is applied to all pointers stored in the registers, caches, and memory.

Displaced Code and Data Pointers. Under address space displacement, attackers must have knowledge of d_{CP} or d_{DP} to forge a code pointer or data pointer, respectively. In Morpheus, these keys are 60 bits in length (see Section 3.3.1.2), providing significantly more entropy than ASLR [5].[1] Displacement is the only protection applied to pointers in the pipeline. Displacement does not obfuscate the relative distance of code or data objects.

[1] ASLR typically has at most 30-bits of entropy, due to architectures only partially implementing virtual addresses (*e.g.*, RISC-V and x86-64 have 48-bit virtual addresses) and because code and data have to avoid collisions when placed.

Thus, attackers may be able to correctly forge pointers by incrementing an existing pointer by an offset. We discuss this limitation further in Section 3.7

3.2.4 Encrypting Implementation Information Assets

Morpheus encrypts three classes of implementation information—code, code pointers, and data pointers—to defend against attacks that inspect or forge these assets. Each value is encrypted under individual domain keys (k_C, k_{CP}, and k_{DP}) using a strong cipher in counter-mode. These values are decrypted before use to maintain program correctness. To reduce the performance overheads of using a strong cryptographic scheme, we only encrypt these assets when they reside in the L2-cache or in memory. Values are decrypted when brought into the L1-cache and encrypted when written back to the L2-cache. Under this design, code is unprotected in the I-cache and pointers are *only* protected by address space displacement while in the D-cache or pipeline.

Encrypted Code and Data Pointers. Malicious injection of a pointer, as exemplified by the stack-smashing exploit in Figure 3.1, requires knowledge of the encrypted representation of that value. Since pointers are always encrypted outside of the pipeline and L1 caches, any attempt to exfiltrate them by writing them to an I/O location or via DMA, RDMA, or cold-boot attack (all of which access DRAM or the L2-cache) will produce an encrypted pointer. Since we use address information in the encryption of pointers, re-injection of this leaked pointer at a different memory location is invalid. Thus, attackers are extremely limited in using encrypted pointers to launch their attacks. Forgery of encrypted pointers would require knowledge of the cryptographic key or manipulation of an existing pointer.

Encrypted Code. Encrypted code thwarts code injection attacks that insert malicious instructions in the data segment (*e.g.*, stack). Further, encrypted code also inhibits the discover of pointer values from analyzing the program binary. An attacker may attempt to subvert pointer encryption by analyzing the executable to learn absolute and relative code locations. By maintaining encrypted code, attempts to read code via I/O are sufficiently obfuscated, further defending against control-flow attacks.

3.2.5 Re-Randomizing Morpheus' Protections with Churn

Address space displacement and domain encryption strip attackers of the implementation information needed to synthesize control-flow exploits. However, even with strong cryptography, attackers can still infer some information about these assets using leakage mechanisms

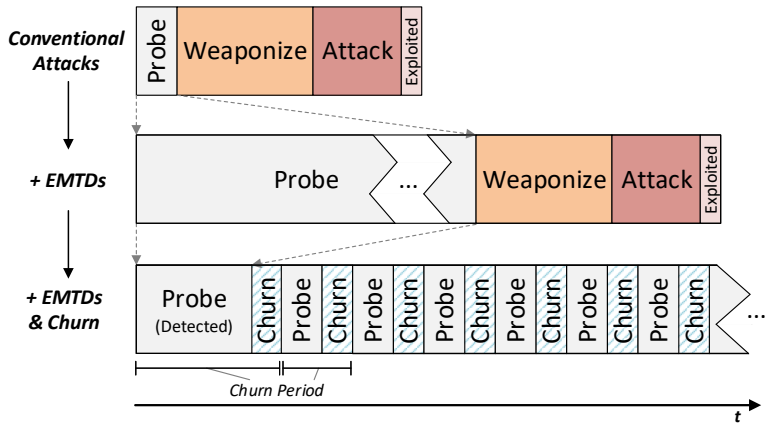


Figure 3.3: **Thwarting Implementation Information Acquisition with Churn.** Exploits progress with the following phases: *probe*, *weaponize*, and *attack*. Ensembles of moving target defenses (EMTDs) increase uncertainty in key values needed to weaponize attacks, thus, probe times increase significantly. To ensure the patient attacker does not complete their probes, churn re-randomizes key program values at regular intervals. An attack detector senses when probe-like activity is occurring and reduces the churn period to strengthen defenses.

like side channels. In Morpheus, code and pointers are decrypted to maintain program functionality, leaking related plaintext information through observable program traces (*e.g.*, memory access patterns and timing). Every time a code pointer is used to redirect program execution, information about the true memory address contained in that pointer is leaked via observable program traces. Likewise, every time an instruction is decrypted and executed, information about the instruction's functionality is leaked via timing and contention in the architecture. Clever attackers could use side-channel analysis or other mechanisms to learn the values of these information assets.

Morpheus Churn. To combat information leakage, we dynamically re-randomize code and pointers under new displacement and encryption keys, termed ***churn***, to nullify any information an attacker may have gained via side-channel leakage. After this re-keying, all instructions and pointers in the system take on a new form, ensuring that any valuable information gathered by attackers since the last churn cycle will be lost. This approach creates high-entropy situations where attackers are confronted with a system they have never seen before and will never see again. Churn can be configured to occur periodically or in response to a particular event, such as a security violation or exception.

As shown in Figure 3.3, conventional attacks probe the system for implementation assets before synthesizing and delivering the attack payload (as discussed in Chapter 2 Section 2.3.2.1). Traditional EMTDs obfuscate these assets at load-time, forcing attackers to use more sophisticated probing techniques to acquire implementation information (*e.g.*, Sec-

tion 2.5.2). Churn re-randomizes these assets periodically during execution. With fast churn periods, attack probes fail to uncover the values for which they search before the system is re-randomized. To make this environment more hostile for attackers, we incorporate an ***attack detector*** to support churn. Detection of suspicious behavior, such as arithmetic on code pointers, triggers an immediate churn cycle, shortening the window for an attacker to synthesize an attack. More details on this attack detector are found in Section 3.3.1.1.

3.3 The Morpheus Secure Architecture

Morpheus implements its domain displacement and encryption protections in hardware to reduce the performance impacts on running programs. In this Section, we detail the implementation of Morpheus on a 64-bit RISC-V in-order core. The Morpheus RISC-V architecture is shown in Figure 3.4. Morpheus' design relies on memory tags to precisely track and identify code, code pointers, and data pointers at runtime. These memory tags enable us to identify memory locations that require re-randomization during churn and decrypt pointers and code correctly at the L1-L2 boundary. Further, the use of memory tags facilitate attack detection, as we can distinguish between data domains in the pipeline to flag suspicious behavior and increase the churn rate in response to a potential attack.

3.3.1 Hardware Architecture

Below, we discuss the implementation details of the Morpheus displacement (Section 3.2.3), encryption (Section 3.2.4), and re-randomization (Section 3.2.5) protections, in that order.

3.3.1.1 Runtime Domain Tagging & Enforcement

We introduce memory tagging to enable the Morpheus architecture to precisely track the domain of each memory object during execution. Morpheus tracks four implementation information domains—code (C), code pointers (CP), data pointers (DP), and other data (D)—applying a 2-bit tag for every pointer-width word. The memory, caches, and register file are extended to store these additional 2-bits for every 64-bit aligned word.[2] Domain tag information is appended to each cache line and register file entry. In DRAM, tag information

[2]To reduce the overhead of memory tags, Morpheus only attaches one tag to each 64-bit aligned word, as pointers in the RISC-V RV64 ISA are 64 bits wide. Since RISC-V instructions are 32 bits wide, we augment the linker to minimally NOP pad each object file so that every 64-bit code location contains two 32-bit instructions.

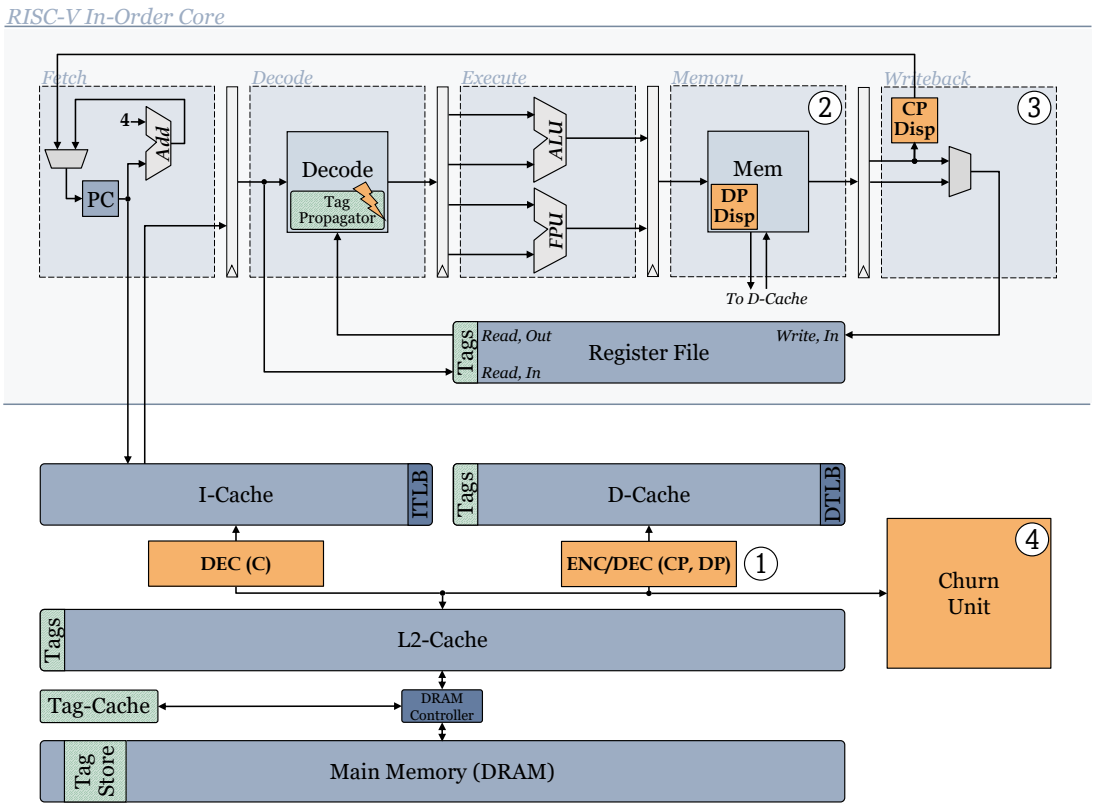


Figure 3.4: **Morpheus Architecture.** The Morpheus RISC-V extensions decrypt code and pointers at the L1-cache boundary ① and correct displaced pointers in the pipeline ②/③. The Churn Unit ④ supports re-randomization during runtime. Churn support also includes the Attack Detector and Threshold Register, which are now shown in this figure. The memory system is extended with 2-bit tags to track the domains of data values at runtime, while the Tag Propagator determines the tags for computed values.

resides in a non-addressable portion of the address space called the *tag store*. By concentrating tags to a fixed location in DRAM, we simplify the churning process as domain types can be efficiently located within the tag store. When data is loaded from DRAM to the caches, a value's tag information is accessed at $tagstart + phyaddr/32$ and appended to the cache line, where *tagstart* is the starting address of the tag store. To improve tag access latency, 64-byte tag groups read from DRAM are cached in a *tag cache* for future lookups.

Tag values are initialized by the compiler, as detailed in Section 3.3.4. The pipeline is responsible for propagating tags. For each instruction, the pipeline computes the tag of the destination register or memory location (`RD`) based on the source register tags (`RS1`, `RS2`) and the instruction's opcode, following the rules listed in Table 3.1.

RS1.tag	Inputs Opcode	RS2.tag	=	Output RD.tag	
C	`ANY`	*X*		N/A	*Code modification results in an exception, per Table 3.2*
X	`COMPARE`	*X*		D	*Compare operations always produce data*
X	`Any`	*Immed.*		RS1.tag	*Operations using immediates propagate the register tag*
CP	`Any`	*X*		CP	*Code pointer operations always produce code pointers*
X	`JAL/R`	*X*		X	*Jumps always produce code pointers*
DP	`SUB`	DP		D	*Data pointer subtraction produces data*
DP	`LD/ST`	*X*		RS2.tag	*Accesses propagate the tag of the value being loaded/stored*
DP	`ANY`	*X*		DP	*All other data pointer operations produce data pointers*
D	`ANY`	D		D	*All other data operations produce data*

Table 3.1: **Tag Propagation Rules.** This table lists the rules used to determine output tags (RD.tag) in the pipeline. These rules work like an if-else tree: the rules at the top are evaluated *first*, and if it does not apply, it moves to the next row in the table. The tag *X* implies that it can have any value and tag *Immed.* indicates that the second operand is an instruction immediate. `RD` is the destination register or destination memory location, in case of `ST`.

Attack Detection. As shown in prior tagged architectures [60, 143, 144, 145, 146], memory tagging offers security benefits through policy enforcement. Morpheus makes use of tag checking policies to provide additional security guarantees via its ***attack detector***. Table 3.2 lists the policies enforced by the attack detector. HIGH-level exceptions are raised for egregious operations that are indicative of malicious behavior. These exceptions terminate the program when raised. LOW-level exceptions are raised when suspicious but potentially benign activity is detected. Many of these exceptions relate to the use of undefined semantics (*e.g.*, testing bits of a code pointer by computing $p\&(1 << n)$). These exceptions trigger a churn cycle to destroy any leaked information that may have resulted from the instruction.[3] For more information on undefined semantics, their use in attacks, and how normal programs may exhibit these behaviors, please refer to Section 2.2.

Although the attack detector could enforce much finer-grained rules, such as prohibiting code pointer arithmetic, we allow these operations to ensure minimal impact on programmers. Use of these suspicious operations result in the added performance penalty of an additional churn cycle, but are not forbidden. In this way, Morpheus does not force programmers to re-write or annotate their code to confirm to its security policies. Since we can deliver increased security without making additional demands on programmers, we have found that our approach is preferred over systems that force programmers to re-write or annotate potentially dangerous code (*e.g.*, CHERI [60]). We have found that normal programs rarely trigger the attack detector, while attack probes trip the detector repeatedly, forcing attackers to contend with additional churn cycles (Section 3.6). If the detector trigger rate

[3]It is important to note that churn is always occurring at a baseline rate in Morpheus, and the attack detector only serves to *increase* the churn rate when a potential attack is detected.

Level	Opcode	Tag Check	
HIGH	`ANY`	Insn.tag != C	*Only code (C) can be executed*
	`ANY`	RS1.tag || RS2.tag == C	No CODE in the pipeline
	`JAL(R)`	RS1.tag != CP	Jump target must be CP
	`LD/ST`	RS1.tag != DP	Memory address must be a DP
LOW	`ANY` (not JAL(R))	RS1.tag == CP	CP arithmetic suspicious
	`ANY`(not LD/ST)	RS2.tag == DP	DP arithmetic suspicious, except add/sub D
	`COMPARE`	RS1.tag != RS2.tag	No compares across domains
	`ANY`	Overflow Occurs	Overflows are undefined
	`SHIFT`	Shift > RegWidth	Invalid shift operand is undefined

Table 3.2: **Attack Detector Logic.** This table shows rules implemented by the attack detector. ABORT rules monitor particularly grievous operations and trigger an exception that terminates the program. CHURN rules detect undefined semantics that may be indicative of an ongoing attack. In response to these violations, the attack detector initiates a churn cycle. Tags are defined as follows: C = Code, CP = Code Ptr, D = Data, DP = Data Ptr.

rises above a concerning level, the attack detector could be configured to raise a signal to the OS to indicate that an ongoing attack is likely.

3.3.1.2 Address Space Displacement

Morpheus obfuscates pointers with 60 bits of random displacement. Specifically, pointers are incremented by the 64-bit d_{CP} or d_{DP} key, where the lower 4 bits are zero to accommodate codes that enforce physical memory alignment on pointers, such as `memcpy`. Pointers are displaced throughout the architecture and in the register file, but are transformed from DAS→VAS before being used for an instruction fetch or memory access. In RISC-V, the effective address of a memory access is formed by adding the instructions offset to a register value. In Morpheus, this address computation must also subtract the current displacement. We pipeline this transformation to reduce exposed latency in the pipeline. In the decode stage, we prematurely subtract the displacement key from the instruction offset ($offset - d_{DP}$). Then, in the execute stage, this value is added to the base register, producing the VAS effective address with no additional latency. A similar approach is used for `JAL/R` instructions. All other instructions are performed on DAS pointers directly. For example, data pointer arithmetic (*e.g.*, `arr[i]`) proceeds on DAS pointers without alteration.

Displacement keys are kept secret from software to prevent direct leakage. Multi-processing execution contexts need only share a displacement if they are sharing code or data, otherwise, each address space will have its own unique displacements. Finally, the kernel has two sets of displacements: one for privileged memory and one for user space.

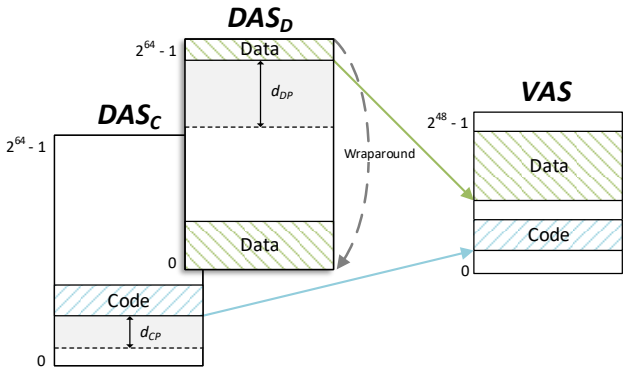


Figure 3.5: **Address Space Displacement.** Address space displacement creates two displaced address spaces, DAS_D and DAS_C, shifted by a random displacement of d_{DP} and d_{CP}, respectively. Pointers are expressed in these superimposed address spaces but translated back to the VAS before use. With two displaced address spaces, any displacement can be used without creating collisions between code and data objects, increasing the entropy for the displacement keys.

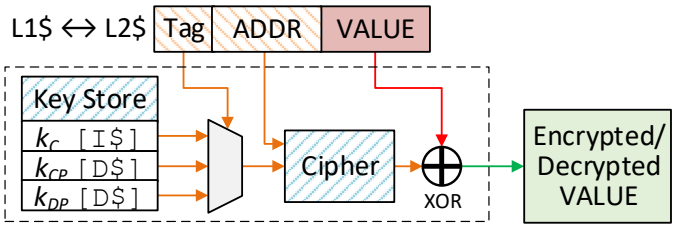


Figure 3.6: **Domain Encryption.** Morpheus' encryption makes use of per-domain keys, denoted as k_C, k_{CP}, and k_{DP} for *code, code pointer,* and *data pointer*, respectively. These keys are selected by a value's domain tag and used to protect it in memory. Non-pointer data is unencrypted.

3.3.1.3 Domain Encryption

Morpheus encrypts code, code pointers, and data pointers under different cryptographic keys (k_C, k_{CP}, and k_{DP}, respectively) while these information assets reside in memory. Specifically, Morpheus decrypts assets when they are brought into the L1-cache and re-encrypts them when they are written back to the L2-cache (*i.e.*, pointers and code in the register file and L1-cache are unencrypted). Morpheus encrypts assets with QARMA [10] in counter-mode using the address as a tweak. We generalize this scheme in Figure 3.6. Our goal of minimizing exposed latencies motivates our choice of encryption scheme and its placement on the L1/L2-bus.

We instrument the processor with two encryption units on the L1/L2-bus, one to handle requests from the I-cache and one to handle requests from the D-cache. These units are specialized for the domains expected from each cache (*e.g.*, only code is valid in the I-cache). During an instruction fetch, the physical address of the fetched instruction is encrypted under

the code key (k_C) while the instruction is read from the L2-cache. When the instruction is returned, it is XOR'ed with the encrypted address. The resultant decrypted instruction is then written to the I-cache. In the case that the value returned from the L2-cache is not tagged as code (C), the QARMA unit interfacing with the I-cache does not perform the XOR and instead stores the value and its tag directly. Any usage of this erroneous value will trip the attack detector and result in a security exception, per the rules given in Table 3.2. This design enables instruction decryption to occur in parallel with the L2-cache access, minimizing delays.[4]

Values stored within the D-cache can be data, data pointers, or code pointers. We design the encryption unit interfacing with the D-cache to accommodate these three domains. Since the tag of the requested value is not known until the L2-cache access returns, the QARMA unit encrypts the requested address under both the code pointer (k_{CP}) and data pointer (k_{DP}) keys. When the value returns, its tag is used to select the appropriate ciphertext to perform the decryption. The decrypted value is then passed to the D-cache to complete the load. If the value is tagged as data (D), it is already plaintext and no XOR is performed. If the value is tagged as code (C), no XOR is performed as any code in the pipeline will trip the attack detector. This design allows us to fully parallelize the QARMA unit logic with the L2-cache lookup, since the L2 access has a latency of 20 cycles and the two QARMA encryptions performed in series has a latency of 18 cycles, per our implementation (§3.4).

By using the effective address as a tweak in the encryption scheme, Morpheus domain encryption protects against cryptanalysis and attempts to learn plaintext values by analyzing the commonality of ciphertexts. Further, this design prevents copy-based attacks that swap memory locations without decrypting their values, as swapping memory locations would result in the wrong address being used for the decryption, producing an incorrect output.

The three keys used for domain encryption (k_C, k_{CP}, and k_{DP}) are stored in microarchitectural registers that are not accessible by software, so they are private to a process's address space. Multi-processing execution contexts need only share encryption keys if they are sharing code or pointers, otherwise, each process will have its own unique keys. Finally, the kernel will have two sets of keys: one for privileged memory and one for user space.

3.3.2 Churning Moving Target Defenses

In Morpheus, the domain encryption and address space displacement defenses are re-randomized at runtime beneath live execution. The ***churn unit*** facilitates this re-keying

[4]A similar technique was used to speed up access to the encrypted instruction cache in the Polyglot architecture [82].

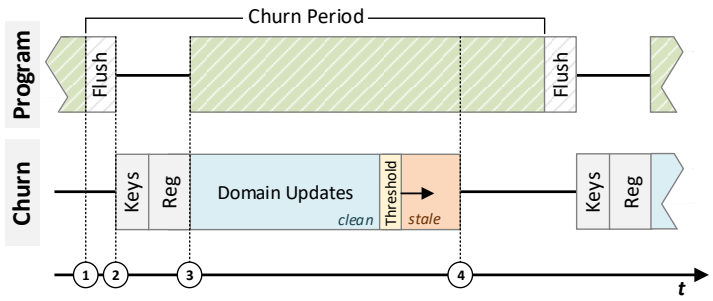


Figure 3.7: **Churning During Execution.** The churn cycle starts with ① a pipeline flush and ② new key generation and register updates. Next, ③ a threshold register is used to coordinate updating values in memory under the new displacements and encryption keys. When ④ churn completes, all domains have been updated, effectively disposing of any information the attacker may have acquired previously.

and updates system state concurrently to minimize performance impacts on running programs. We implement the churn unit as a simple hardware co-processor with access to the L1/L2 cache-coherent bus and main core to facilitate memory and register updates.

When a system churn is initiated, either because the current churn period has elapsed or the attack detector is triggered, the churn unit signals for a pipeline flush (① in Figure 3.7). This action ensures that any in-flight instructions are completed before new keys are introduced. While the pipeline is halted, a new set of domain encryption and displacement keys are generated (*e.g.*, k_{CP}^{new}, d_{CP}^{new}) and a process context switch is initiated ②. This context switch is used to update the register file to match the new keys. When the context is switched out, the stale register contents are written to memory. When restored, the register contents are populated from memory via load instructions, which automatically update values to match the new keys, as detailed below. After instruction fetch resumes, the churn unit walks the virtual address space of the active process from low to high addresses and updates code and pointers through requests to the L1-to-L2 bus ③. Specifically, the unit reads in a cache line. If the line is flagged as "dirty" in the data cache, it is written back and invalidated from the L1 before being read into the churn core. For each code pointer in the cache block, the pointer is decrypted under the old encryption key (k_{CP}^{old}), incremented by the difference between the displacements ($d_{CP}^{new} - d_{CP}^{old}$), and re-encrypted under the new encryption key (k_{CP}^{new}). Updates to data pointers and instructions proceed similarly, except displacement is not applied to instructions. To prevent race conditions between the churn unit and kernel, virtual memory pages are locked while being updated.

Since re-keying occurs during program execution, protected domains exist in memory in both a "stale" (protected under the old keys and awaiting update) and "clean" (protected under the new keys) state. The pipeline and Morpheus defenses are configured to operate

correctly under this mixed state. First, the churn unit communicates its progress to the pipeline via a ***threshold register*** that indicates the current virtual address being updated. Any application memory requests or instruction fetches below the threshold will retrieve clean values into the pipeline. Any requests above the threshold will retrieve stale values from memory, which will be updated to clean before entering the pipeline. If requesting access to the memory location represented by the threshold register, the pipeline must stall until the churn unit finishes its update.[5] These invariants are summarized as follows:

i) Memory values *below* the threshold address are clean

ii) Memory values *above* the threshold address are stale

iii) Memory values *at* the threshold address (cache line granularity) are currently being updated

iv) The pipeline and register contents are always clean and values are updated to use the new keys upon entering the pipeline

The following paragraphs detail how instruction fetch and memory accesses are modified to uphold these invariants during churn.

Maintaining Correctness for Mixed State During Fetch. The I-cache is decrypted. When an instruction is brought into the I-cache, the instruction's effective address (in VAS) is compared to the threshold register and used to select the new or old keys. Decryption of the instruction proceeds as described previously using the appropriate set of keys.

Maintaining Correctness for Mixed State During Loads/Stores. When data is brought into the L1-cache or written back to the L2-cache, the memory request's effective address (in VAS) is compared to the threshold register and used to select the new or old keys. Decryption or encryption of the requested value proceeds using this set of keys. This process is similar to maintaining invariants (i) and (ii) in the I-cache.

Address space displacement requires additional modifications to maintain invariant (iv) when bringing pointers into the pipeline. If a load with an effective address *above* the threshold returns a pointer, this pointer must be adjusted to reflect the new displacement before being written to the register file. Loads to effective addresses *below* the threshold require no updates since the pointer and destination are both clean. Similarly, when a store

[5]This requirement prevents write-after-read and write-after-write hazards that could occur when the core and churn unit access the same address.

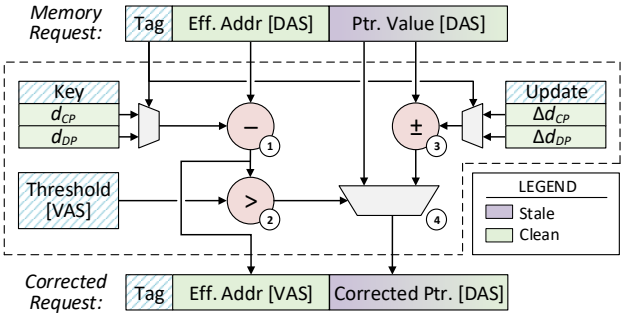


Figure 3.8: **Address Displacement Gate.** ① Regardless of if the system is churning or not, we always remove the most up-to-date d from the effective address before accessing the TLBs, since the pipeline will only generate requests with the newest d. ② When churning, the gate must determine if the value being loaded/stored is within a region yet updated by the Churn Unit or not. ③ If the pointer is being *loaded*, then it must be made "Clean" by adding Δd. On *stores*, the pointer may be made "Stale" by removing Δd. ④ Finally, if the pointer comes from the "Clean" region, the gate selects the value as-is without modification.

writes a pointer to memory *above* the threshold register, the clean pointer in the register must be updated to the old displacement to maintain invariant (*ii*). Stores to effective addresses *below* the threshold require no updates since the pointer and destination are both clean. We amend the address space displacement mechanism to update these pointers accordingly, as shown in Figure 3.8.

Continuous Churn. Once the churn unit has walked the entire virtual address space of the current process, all code and pointers have been updated to reflect the new keys ④. The churn unit then sleeps until another churn cycle is initiated. If the churn process immediately restarts ①, we regard this as "continuous churn." The system's churn period and attack detection can be configured to meet the desired security-performance trade-offs. We analyze the performance impacts of various churn rates, ranging from 200 ms to continuous, in Section 3.5.

3.3.3 Support for Context Switching

Morpheus must interact with the operating system (OS) for routine tasks, like context switching. To prevent adverse impacts to OS scheduling, the churn unit supports context switching during an active churn cycle. The OS can request the current context from the churn unit, which is encrypted with a boot-time private churn key. The encrypted context is passed to the kernel, which stores it in the kernel's process control block. The churn context contains the threshold register, the old keys for all defenses, the new keys for all defenses, and the

time that the last churn cycle was initiated. To prevent attacks on mostly idle programs, the churn unit initiates a churn cycle on a program after it has been idle longer than its normal churn cycle (as indicated by the timestamp in the churn context).

3.3.4 Compiling Protected Applications.

Applications must engage Morpheus' protections via the compiler. The Morpheus LLVM-based [147] compiler extensions provide the locations of code and pointers in the executable to enable precise tracking of domains at runtime. This information seeds the domain tagging infrastructure and allows the precise tracking of domain tags at runtime. The input to the compiler is unmodified C source files, which are converted into the LLVM intermediate representation by the Clang front-end. Then, a global variable domain analysis labels each memory object in statically initialized data sections as data, a code pointer, or a data pointer. Subsequently, an instruction labeling pass in the Clang back-end identifies and labels instructions that initialize dynamically created memory objects (*i.e.*, values on the stack, heap, and `.bss` segment). Finally, a Python-based parser extracts initial domain tag information from the labeled binary. The output of the compiler is a binary with labeled instructions and a domain tag file that contains the initial tags for memory objects.

3.4 Experimental Methodology

To assess security and performance, we implemented our Morpheus prototype on the RISC-V port of the gem5 simulator [6, 148]. The Morpheus core is built on top of the gem5 MinorCPU 4-stage in-order core using the configuration parameters listed in Table 3.3. We chose this core model because it would demonstrate the impact of EMTD protections for a core with little capability to tolerate any introduced latencies (as opposed to an out-of-order core). Morpheus' encryption layer uses QARMA$_7$-64σ_1, which, when synthesized in [10], had a minimum delay of 3.25 ns (9-cycle latency at 2.5 GHz). QARMA$_7$-64-σ_1 is a fast, lightweight encryption algorithm that encrypts 64-bit blocks with a 128-bit key in 16 rounds. The latency of using QARAMA in counter-mode for domain encryption is masked by the latency of the 20-cycle L2-cache access. We used DRAMSim2 [149] to model the memory system and assess the performance of tag scanning and churn operations. Specifically, the churn unit communicates between gem5 and DRAMSim2 to scan tags and update pointers and code. The simulations model a single process context, using the system call emulation capabilities of gem5.

Core Type	2.5GHz, MinorCPU (In-Order)
Cache Line Size	64B
L1 Cache	32KB with 2-cycle latency
L2 Unified Cache	256KB with 20-cycle latency
Tag Cache Size	4KB

Table 3.3: **Experimental Configuration for gem5.** The specifications of our baseline RISC-V in-order core in gem5 [6]. A 4KB tag cache is added to the EMTD-enabled microarchitecture.

Compiler Support and Benchmarks. To assess the performance overheads of EMTD defenses, we ran experiments on our augmented gem5 simulator with all Morpheus protections engaged (*i.e.*, domain encryption and address space displacement). We evaluated Morpheus using benchmarks from MiBench [150] and SPEC CPU2006 [151, 152]. By comparing the performance impacts on these two suites, we can get an understanding of the relative costs for small embedded programs (MiBench) and larger desktop/server workloads (SPEC CPU2006). Simulations were run with a 3 billion instruction cap. From MiBench, we analyzed the performance of 13 programs: *basicmath*, *bitcount*, *qsort*, *susan*, *dijkstra*, *patricia*, *stringsearch*, *blowfish*, *rijndael*, *sha*, *crc32*, *fft*, and *adpcm*; and from SPEC CPU2006, we analyzed the performance of all C-code benchmarks, 12 in total, using the reference input: *perlbench*, *bzip2*, *gcc*, *mcf*, *milc*, *gobmk*, *hmmer*, *sjeng*, *libquantum*, *h264ref*, *lbm*, and *sphinx3*.

All programs were built using LLVM 5.0.0 for the RISC-V RV64IMA architecture [153] with optimization level -O2. The benchmarks were linked against a Morpheus-built RISC-V Musl C library [154].

3.5 Experimental Results and Evaluation

To measure the performance impacts of Morpheus, we ran experiments against the MiBench [150] and SPEC CPU2006 [151, 152] workloads for various churn rates. This analysis serves to demonstrate how churn's memory updates affect performance on top of the slowdowns imposed by encryption latencies. We also estimate the area overheads of the Morpheus defenses. These experiments are discussed below.

3.5.1 Runtime Performance Overheads

Figure 3.9 shows the slowdown for the MiBench and SPEC CPU2006 benchmark suites with full Morpheus protections at varied churn periods. The slowdowns measured are with

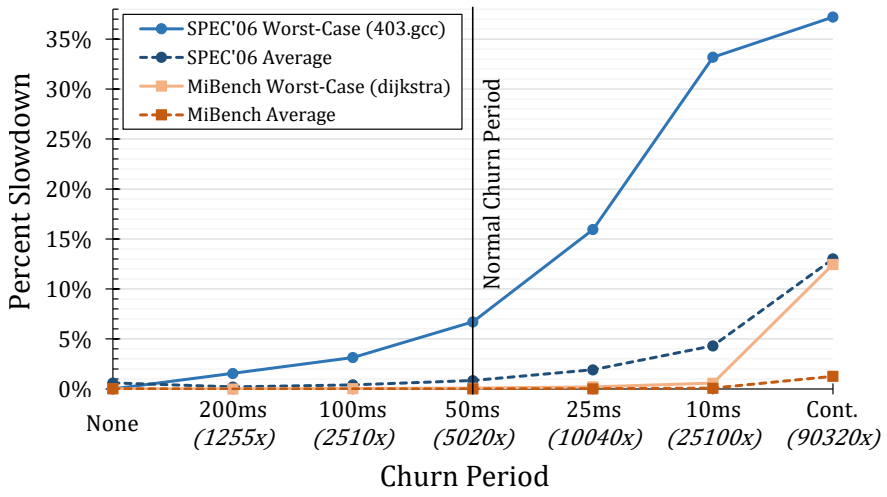


Figure 3.9: **Performance Overhead.** This graph shows the average and worst-case performance overheads of the chosen MiBench and C-based SPEC CPU2006 benchmarks, with churn periods varying from no churn (left) to continuous churn (right). The expected number of churn periods that would complete in the time estimated to penetrate Morpheus defenses, based on the attack study in Section 3.6, is shown below each churn period. Given these results, we target 50 ms as a "normal" churn period.

respect to the program running on the baseline architecture without Morpheus protections. For both MiBench and SPEC CPU2006, the graph shows the average slowdown at each measured churn period, plus the benchmark with the highest slowdown (*i.e.*, *dijkstra* for MiBench and *gcc* for SPEC CPU2006). The points on the far right of the graph represent a continuous churn period, where a new churn cycle starts immediately after the previous one ends. For these points, the churn period varies by program; the average continuous churn period for SPEC CPU2006 was 2.8 ms, with *gcc* having the slowest churn time of 19.4 ms. For MiBench, the average churn time was much smaller, at 0.4 ms, with the worst case being *patricia* at 2.1 ms.

As shown in Figure 3.10, slowdowns for short churn periods are quite low, with an average overhead of 0.84% (and a worst-case slowdown of 6.71%) at 50 ms. Overheads even seem tolerable up to 10 ms, but quickly grow as the system approaches continuous churn. These low overheads are the result of efficient hardware-based churn, and the fact that code and pointers only make up a small fraction of overall memory. Non-pointer data is not accessed during the churn phase. Surprisingly, running without churn results in a slightly higher overhead than running with 200 ms churn (0.6% vs. 0.2%), because our churn unit is essentially acting as a cache prefetcher. To better understand the churn unit's workload, we analyzed *gcc* since it had the largest slowdown. We found that *gcc* has *i)* The largest data segment, with an average footprint of 47.8 MB, *ii)* The most pointers, with over 300,000,

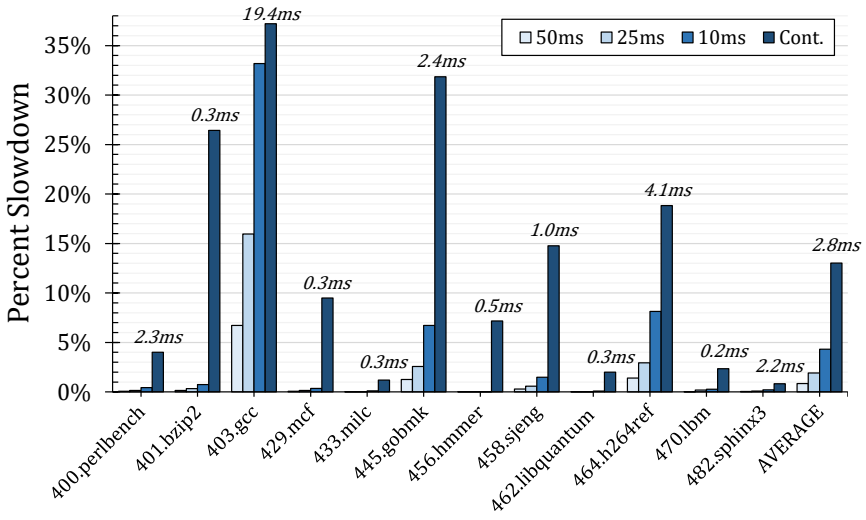


Figure 3.10: **SPEC CPU2006 Performance Overheads.** The graph shows SPEC CPU2006 performance overheads for varied churn periods. The continuous churn period is listed above the bar and differs based on the time needed to update all memory values used by the workload.

and *iii)* The largest code base, at 5.3 MB. The combination of these factors resulted in more work for the churn unit, and therefore higher overheads.

It is interesting to consider how fast the churn period should be. In existing network security protocols, churn periods are very large. The re-key period for the TLS secure web protocol defaults to 1 hour (3.6 million ms) [155]. In Section 3.6, we find that it took 251 sec to penetrate the de-featured Morpheus architecture. The number of churn cycles that would complete in that time is shown in parenthesis on the x-axis of Figure 3.9. Here, we see that ***a churn period of 50 ms is 5020× faster than the expected attack time, thus we see 50 ms as a reasonable "normal" churn period.*** In all likelihood, an attack would frequently trigger the attack detector, leading to continuous churn, which is on average 90,320× faster than the expected attack time.

3.5.2 Area Overheads

The hardware overhead of Morpheus is estimated with relation to the baseline core, SiFive's U54-MC. Table 3.4 summarizes the parameters of the baseline core found in [7]. The table also lists the area estimates for caches produced by CACTI 7 [9], using the parameters for the U54-MC. We acquire estimates for the QARMA block cipher from [10]. Since the original paper uses the 7nm process, we scaled their estimated area to the 32nm node (to remain conservative), using scaling factors from a VLSI journal article [11]. The E51 [8] was selected as a conservative estimate for the churn unit, as it is a 64-bit core with more features than a

Baseline Architecture		Morpheus Architecture	
Component	Area (mm^2)	Component	Area (mm^2)
1x U54 Core	0.234	1x U54 Core	0.234
L1 D$: 32KB	0.332	L1 D$: Tagged	0.351
L1 I$: 32KB	0.332	L1 I$	0.332
L2: 512KB	1.351	L2: Tagged	1.397
		Tag$: 4KB	0.019
		1x E51 Core (Churn Unit)	0.082
		3x QARMA Units (Scaled)	0.044
Total Area	2.249	Total Area	2.459
		Morpheus Overhead	**9.34%**

Table 3.4: **Morpheus Hardware Estimation.** Numbers for SiFive cores were obtained from their website, for the U54 [7] and the E51 [8]. Caches are estimated using CACTI 7 [9]. QARMA units have a size of $1879.9\mu m^2$ in 7nm [10]. This is scaled by 7.8 (as given in [11]) to get $0.0147mm^2$ per unit in 32nm technology.

specialized churn unit would need. We also use 3 QARMA units in our estimation to account for the L1 I$ path, L1 D$ path, and one within the churn unit to accommodate re-keying. We do not include numbers for pipeline modifications, such as the tag propagation unit, as that area is negligible compared to the other additions.

Of the 9.34% overhead incurred by Morpheus, a third of it accounts for memory to accommodate tags. The biggest component is the churn unit, which can be reduced by designing the churn unit as a more specialized hardware unit, instead of using a general-purpose core.

3.6 Security Analysis

To gauge the security benefits of Morpheus, we perform a simulation-based study of its ability to stop a wide range of control-flow attacks, ranging from long-standing attacks (*e.g.*, buffer overflow) to advanced attacks (*e.g.*, return-oriented programming). We then attempt to build Morpheus-specific attacks, to assess the value of ensembles and churn.

3.6.1 Penetration Testing

To build confidence that the Morpheus architecture is capable of stopping control-flow attacks, we performed penetration testing with real-world control-flow attacks running on the simulated Morpheus architecture. We ran tests from an ongoing port of the RIPE control-

Attack	Defenses Engaged					Bits of Entropy
	E-C	E-CP	E-DP	P-CP	P-DP	
Stack Buffer Overflow [54]	X	✓	X	✓	X	188
Heap Overflow [156, 157]	X	✓	X	✓	X	188
Heap Spray [66]	X	✓	X	✓	X	188
Format String [158, 159]	X	✓	✓	✓	✓	376
Integer Overflow [160]	X	✓	✓	✓	✓	376
ROP [55]	✓	✓	✓	✓	✓	504
Back-Call-Site Attack [62]	✓	✓	✓	✓	✓	504

Table 3.5: **Penetration Testing Results.** The Morpheus secure architecture was shown in simulations to stop all of the attacks listed. Additionally, all attacks trigger our attack detector. For each attack, the table shows how Morpheus stops the attack, with Morpheus domain encryption (E) and pointer displacement (P) applied to code (C), code pointers (CP), and data pointers (DP).

flow attack suite [161]. The port, being developed by Draper Labs [162], does not yet implement the entire RIPE attack suite, but it did give us access to stack overflow [54], heap overflow [156, 157], and ROP [55] attacks. In addition to RIPE, we included hand-crafted implementations of additional attacks, including heap spray [66], format string [158, 159], integer overflow [160], and back-call-site [62] attacks. All attacks aimed to overwrite an existing return address or function pointer as a means to manipulate control flow.

The Morpheus architecture stopped all of the attack classes from our penetration testing suite. Table 3.5 lists the attacks tested and which Morpheus defenses aided in stopping the attack. Stopping a comprehensive control-flow attack suite shows that the Morpheus architecture is capable of stopping state-of-the-art control-flow attacks. Sophisticated software protections that come close (e.g., CPI [76]) are significantly more expensive and advanced hardware protections such as the soon to be released Intel CET [163] extensions have notably lower coverage for advanced attacks.

Of the control-flow attacks in Table 3.5, one of particular note is the *Back-Call-Site* attack, which is a recently published control-flow attack designed to circumvent advanced CFI-based control-flow protections [62]. Morpheus stops this attack when it attempts to forge a code pointer to redirect a function return. It is interesting to note that ***we learned of this last attack after our Morpheus architecture was designed, suggesting that Morpheus provides some level of future-proofing against unknown future CFG attacks***.

Table 3.5 shows that the number of defenses stopping the attack grows as attacks become more advanced. This growth in the effective strength of Morpheus occurs because new attacks nearly always need to acquire additional information to circumvent defenses introduced to stop earlier attacks. For example, buffer overflow attacks [54] do not need knowledge of

program code. However, the introduction of non-executable memory led to code reuse attacks like ROP [55], that do need knowledge of the program code. ***Our analysis suggests that, since Morpheus defends critical attack assets rather than patching vulnerabilities, Morpheus is more effective even as attacks become more advanced.***

3.6.2 Attacking Morpheus

We can gain a deeper understanding of Morpheus' security benefits by crafting new attacks designed to penetrate Morpheus defenses. Our attack scenario involves a local program attacking a victim program (*i.e.*, SPEC's *gobmk* in our study) via an IPC interface, where the program under attack has exception reporting and crash recovery comparable to vanilla Linux. The goal of the attack is to call the C-library `system()` function, thereby giving an attacker a shell in the victim program's process context. We assume that the organization of the code has been randomized at build time (similar to [82]), and the victim program is crash-resilient.

Crafting successful attacks to penetrate Morpheus required degrading its protections in three ways: *i)* Churn and the attack detector are disabled to give the attacker unlimited time to probe the system, *ii)* A single displacement key is used across all address spaces, since microarchitectural sharing attacks are not possible with different displacements and *iii)* A single key for domain encryption is used to allow code pointer forgery.

Brute force attacks in this context are infeasible due to the large randomization space of Morpheus defenses, leading attackers to use side channels to leak information. When attacking Morpheus, we take this approach by combining techniques from the AnC derandomization attack [25] and blind code search [24]. We simulated all aspects of each attack described below, except for IPC calls, exception handling, and AnC derandomization. Instead, reported times were used [25, 164, 165]. Being conservative, we assume AnC can recover all 48-bits of the virtual address in 150 s, whereas the original work recovers fewer bits due to limited entropy in the experimental setup.

As shown in Figure 3.11, we evaluate the time it takes to attack the de-featured Morpheus for each of the four combinations of moving target defenses: $\overline{EP}$, with no defenses, $E\overline{P}$, with only encryption, $\overline{E}P$, with only address space displacement, and EP, with all defenses engaged.

$\overline{EP}$: To execute a call to `system()` when no defenses are engaged, an attacker needs to simply search the victim's code to locate this function. This can be done through exploiting a stack buffer overflow to call `memchr()`, which returns the address of `system()` in 10 ms.

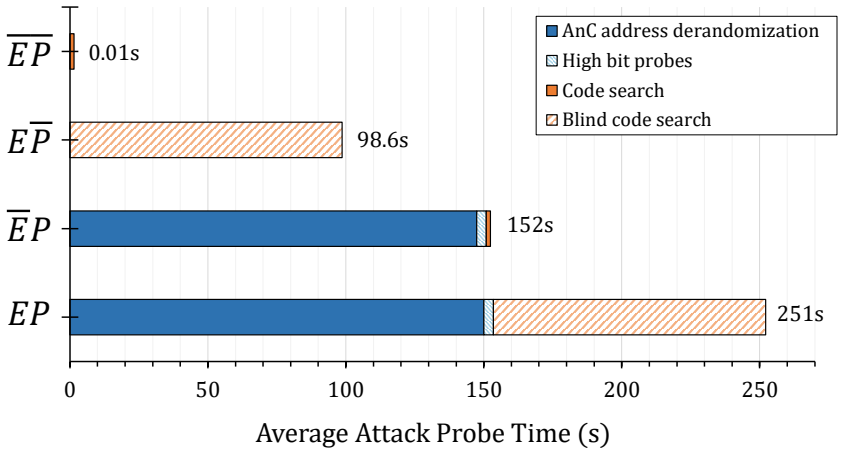


Figure 3.11: **Estimated Probe Times.** This figure shows the average estimated attack time to penetrate a de-featured Morpheus, which has fewer defense keys and no churn. Probe times are shown for Morpheus with (E) and without ($\overline{E}$) domain encryption, and with (P) and without ($\overline{P}$) pointer displacement. As more defenses are engaged, probe times grow, with fully-activated defenses requiring a probe time 5020x longer than the normal churn period (50 ms).

***E$\overline{P}$*:** When encryption is engaged, this technique is not applicable because functions like `memchr()` would incorrectly compare to encrypted code. Rather, an attacker can simply make blind calls [24] into the code segment, until successfully calling `system()`. We can make these calls using mimicry [166] to form the code pointer, and use a stack buffer overflow to test the pointer. This process takes about 98.6 sec to identify `system()` and effectively defeats the encryption defense. It should be noted that these operations would trigger the attack detector on fully featured Morpheus.

***$\overline{E}$P*:** When address space displacement is engaged, an attacker needs to determine the displacement key to locate the code segment before searching for the `system()` function. To leak a pointer and uncover the key, an attacker can use an AnC derandomization probe to reveal a virtual address, followed by high-bit probes to determine where the code resides among the remaining 2^{16} possible locations (*i.e.*, the DAS address). Breaking displacement takes 150 sec for AnC and 2.3 sec for high bit probes. After the address space displacement defense is defeated, an attacker can perform a normal code search as described above. Overall, this exploit takes about 152 sec.

***EP*:** Finally, with all defenses engaged, the above techniques are combined to first derandomize address space displacement, and then blindly search the code segment for a call to `system()`. This approach uses a combination of an AnC derandomization probe to recover the 48-bit virtual address, high-bit probes with blind calls to recover the upper DAS address,

and a full blind-call search for `system()`. Combined, this attack takes a total of 251 sec.

It is clear that ensembling defenses carry much value, since attack times progressively increase as more moving target defenses are engaged. Moreover, the time needed to attack the fully engaged defenses is 251 sec. This provides a useful metric for assessing churn periods, which should be much faster than 251 sec. With churn on, attackers resort to brute-force guessing, which will take substantially longer.

When examining the individual attack details, it is possible to see the direct benefits of ensembles. Clearly, with all defenses engaged, it is the worst of both worlds for the attacker, since they do not know where the code is located in the displaced address space, and once found, they cannot inspect it to easily find the location of `system()`. It is also interesting to note that the increase in attack time is purely additive, since, for this attack, defeating one defense provides no value in penetrating the other defense.

3.7 Discussion and Limitations

Morpheus encrypts the representation of code and pointers in memory and displaces code and data locations throughout the architecture. By combining these two MTDs with churn, malicious attempts to learn these assets for use in an attack are restricted via a tight 50 ms time limit. We have shown that these defenses are effective at thwarting control-flow attacks. However, the instantiation of these defenses have limitations.

Encryption targets information leakage of code and pointers, but does not stop a control-flow attack directly. If an attacker were somehow able to create a pointer within the pipeline or D-cache (*e.g.*, inject a data value that is used for a pointer computation), they would be able to completely bypass the domain encryption layer. Further, address space displacement becomes trivial to bypass because it only obfuscates the absolute location of the data and code segments. The relative ordering of objects within a segment are maintained. Thus, if an attacker were to influence the offset of a pointer value, they would be able to do so knowingly, as $A_{das} + offset$ yields $A_{vas} + offset$. We address this latter deficiency in Chapter 4. Ultimately, it is Morpheus' *ensemble* that gives strength against attacks.

Manipulation of domain tags is possible, but is not useful for security attacks. Transferring data between the code, pointer, and data domains still requires knowledge of the displacement and encryption keys. Without this knowledge, the architecture will decrypt (or not decrypt) data under a different key than it was encrypted under, producing a garbled and unusable result. Manipulation of domain tags could be used to subvert the attack detector, however. We note that the attack detector is not essential for security, since it is used to establish

	Name		Assets C	Assets CP	Assets DP	Entropy/Key Size	Runtime Churn	Avg. Overhead
Displacement	64-bit PaX ASLR [5, 167]		-	✓	✓	29-30 bits (48-bit vaddr)	No	3.6%
	TASR [57]		-	✓	✓	29-30 bits (48-bit vaddr)	On I/O	30-40%†
	Remix [59]		-	✓	X	20 bits for Apache on x86	Random Interval	2.8%
	RuntimeASLR [58]		-	✓	✓	28-48 bits (48-bit vaddr)	On `fork()`	0.5%
Encryption	PointGuard [28]		X	✓	✓	64 bits (weak XOR cipher)	No	10.0%
	CCFI [85]		X	✓	✓*	128 bits (strong cipher)	No	23.0%
	ASIST [81]		✓	✓*	X	32-128 bits (weak XOR cipher) or 32 bits (weak transposition)	No	1.0%
	Polyglot [82]		✓	X	X	163 bits (strong cipher)	No	4.6%
Enc. + Disp.	Shuffler [12]	Displacement	-	✓	X	27 bits (48-bit vaddr)	Fixed Interval	14.9%
		Encryption	X	✓*	X	64 bits (weak XOR cipher)		
	Morpheus	Displacement	-	✓	✓	60 bits per segment	Fixed Interval	0.9%
		Encryption	✓	✓	✓	128 bits per asset (strong cipher)		

Table 3.6: **Comparison Against Prior Works.** The assets protected by each work is listed. A ✓* indicates protection for only return addresses (for CP), or frame pointers (for DP). Displacing the code segment is listed as protection for code pointers (CP), rather than for code (C). For Fixed Interval systems, the average overhead is quoted at 50 ms. †TASR's overhead is originally 2.1%, however, as noted in Shuffler [12], TASR uses -Og, which can slow down SPEC benchmarks by ∼30% compared to -O2.

the churn rate. Our suggested normal churn rate of 50 ms should be sufficient for thwarting most attacks even if the attack detector were offline.

The Morpheus does not support self-modifying code or just-in-time (JIT) compilation. Support for these features would require a secure method for issuing code (C) tags at runtime. The existing tagging system also limits the programmers ability to cast between the data pointer (DP) and data (D) domain. Finally, address space displacement implies that any logic operations on pointers, such as shifts, `AND`, or `OR`, will behave differently and may produce a different value that expected.

3.8 Related Work

Networking and software security has adopted many forms of moving target defenses (MTDs), including popular examples like randomizing network-service ports and ASLR. However, prior research typically focuses on protecting a single asset due to the high overhead introduced by applying multiple MTDs. Morpheus supports multiple MTDs in hardware to provide protection for more assets with lower overhead. A comparison of Morpheus against prior works is shown in Table 3.6. In this table, we show the information assets protected by each work, their associated entropies, average overhead, and if they support runtime churn.

Displacement. ASLR [5] randomizes the base addresses of code and data segments each time a program is loaded, but is susceptible to insufficient randomness [24, 89] – worse, a single address leakage through a memory disclosure vulnerability can reveal the location of all code and data [23, 25, 26, 168]. Kil *et al.* [78] proposed a finer-grained permutation of procedures and data objects for ASLR, such that the leakage of a single address does not immediately reveal the location of all program information. Remix [59], TASR [57], and RuntimeASLR [58] augment ASLR with a run-time component either by rearranging direct calls, direct jumps, and indirect jumps (leaving indirect calls intact and hence being vulnerable to return-to-libc attacks [37]) or by re-randomizing the memory layout after sensitive system calls.

Encryption. PointGuard [28] and related techniques [169] have employed encryption to obfuscate pointers as a means to defend against control-flow attacks. Besides having higher overheads, these approaches have consistently been compromised due to either weak encryption (typically XOR-based), or read attacks that extracted the key. In contrast, Morpheus uses strong encryption and encrypts disparate domains (an idea in [170]) with distinct keys held in protected registers.

Encryption has also been used to protect code, as in binary-translation based Instruction Set Randomization (ISR) [72, 73, 171] – with ∼10% overhead common for network-facing services and up to 75% for databases. This overhead decreases to ∼1% with hardware support in ASIST [81]. Randomization schemes have also been employed for data [172, 173], in which static analysis is used to partition memory into classes that each have their own distinct, random mask. Its average slowdown for SPEC is 14%. While these techniques are appealing as they obscure code and data efficiently, they use weak encryption, both from a brute-force protection and cryptanalysis standpoint. As such, Morpheus forgoes these approaches and adopts a strong cipher.

Another approach to program randomization is N-version systems [174, 175, 176], which require an attacker to break defenses in multiple domains simultaneously. Overheads range from 28%-129% for latency and ∼50% for server throughput. While these techniques are powerful, Morpheus defends against the same attacks with lower overheads.

Comparison with Shuffler. The value of hardware-based EMTDs comes into focus when we contrast Morpheus to a software-based moving target defense. The fastest previous continuous address space reorganization technology we are aware of is Shuffler [12], which moves code periodically in the address space of a running program. Shuffler also encrypts return addresses with XOR.

There are some advantages Morpheus has over Shuffler: *i)* Hardware enables faster churn periods, *ii)* More randomization with a total of 504 key bits and strong encryption, *iii)* More domain coverage, including code and code & data pointers, and *iv)* Lower overhead at a 50 ms churn period – <1% compared to 15% (with a worst-case of 45%). An advantage of Shuffler is that it destroys relative distance between code objects, whereas Morpheus' displacement does not. However, Morpheus' hardware-based defenses have lower overhead while delivering more randomization.

Tagged Architectures. Secure architectures often utilize tags to provide defenses with more information at runtime. For example, Dynamic Information Flow Tracking (suhSecureProgramExecution2004) [146, 177, 178, 179, 180, 181, 182] enables analyses to assign labels to sensitive data, code, or inputs. These labels are tracked as the program executes and raise an exception if a 'tainted' label is used for a sensitive operation (*i.e.*, as an information flow 'sink'). This approach is quite powerful at ferreting out vulnerabilities; however, finding all of the vulnerabilities would require *i)* Full path coverage of the program, and *ii)* A very detailed vulnerability model. Both of these challenges ultimately keep detection of all vulnerabilities beyond reach. Our EMTDs, on the other hand, are vulnerability-agnostic, since they hide the information needed to exploit vulnerabilities.

Other architectures, such as CHERI [60], lowRISC [144], the Dover processor [145], and PUMP [143], have implemented tags in hardware to enforce security policies. Like Morpheus, these systems rely on precise tagging of memory objects. However, these systems often face high false-positive rates, leading to the failure of benign programs. Morpheus operates differently, as false-positive security violations from the attack detector only trigger a churn cycle.

3.9 Chapter Summary

While traditional security protections work to find and fix every last vulnerability, EMTDs with churn take the approach of protecting a system by randomizing the information assets that attackers need to craft successful attacks. In this chapter, we presented the Morpheus secure architecture, which brings together multiple moving target defenses to protect a system from control-flow attacks. Address space displacement and domain encryption work in tandem to protect the code and pointers needed to launch control-flow attacks. A hardware-based churning mechanism is able to re-randomize these values at runtime, without impacting normal programs. Together, these protections demonstrate a high level of protection against control-flow attacks with very low overheads.

CHAPTER 4

Strengthening Code Pointer Defenses via Fine-Grained Randomization

Morpheus encrypts code pointers but fails to obfuscate the relative distance between code objects. Thus, attackers may still be able to knowingly manipulate code pointers (*e.g.*, by adding a constant offset) to commit a control-flow attack. Specifically, this corrupted code pointer can effectively redirect program execution to attacker-injected code or existing code gadgets. To further insulate against this attack vector, we employ a Displaced and Dilated Address Space (DDAS)—a novel address space inflation mechanism to obfuscate code pointers, code locations, and the relative distance between code objects. Specifically, DDAS injects functionally-defined "traps" within the code segment to minimize the likelihood that a manipulated code pointer will reach a valid target. With DDAS, an incorrectly forged code address will result in a security exception with a probability greater than 99.996%. Further, by leveraging custom hardware, we are able to achieve a defense with performance overheads well below 5% and similarly low power and silicon area overheads. Using hardware-based address obfuscation, DDAS provides significantly higher entropy at lower performance overheads than previous software techniques, and our re-randomization mechanism offers additional protections against possible pointer disclosures.

The research presented in this chapter appears in:

[32] Lauren Biernacki, Mark Gallagher, Valeria Bertacco, Todd Austin. ***"Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces,"*** 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2020. DOI: 10.1109/HOST45689.2020.9300273.

4.1 Introduction

Many prevalent security exploits are underpinned by precise knowledge of the memory layout of the victim machine. Control-flow exploits are some of the most dangerous, and most common, attacks of this nature [139]. In these exploits, attackers leverage knowledge of the location and representation of code pointers, abusing a memory corruption vulnerability to overwrite control data with malicious pointer values and enable the execution of arbitrary, malignant code on the victim machine [55, 67, 68]. Attackers without knowledge of the memory configuration employ techniques to readily discover this information at runtime [23, 24, 91]. This class of exploits occurs on the *control plane*, as attackers directly manipulate control data and require knowledge of the code segment and the representation of code pointers to be successful.

To deprive attackers of the information needed to synthesize an exploit, protections have employed *address obfuscation*. These defenses randomize the memory layout of the victim machine, effectively prohibiting attackers from accessing memory and forging pointer values at whim. There exists a significant body of work on address obfuscation defenses, yet the techniques used to randomize memory vary broadly. Address obfuscation techniques include address space displacement [5, 31, 57, 58], permutation of code at the function [12, 78, 183, 184, 185], basic block [59, 186, 187, 188], or instruction level [79, 80], insertion of padding between code objects [189], duplication of the code segment [90, 91], and insertion of traps within the code space [190].

While obfuscation based protections can be both effective and efficient, they offer probabilistic security. Thus, achieving both high entropy and fine-grained, frequent randomization is critical for resilience [191]. Memory disclosures, or the leakage of memory addresses, can be readily used to derandomize the address space. For example, with techniques like Address Space Layout Randomization (ASLR) [5] that employ address space displacement, the entire code layout can be inferred from a single pointer disclosure. Even finer-grained code randomization techniques that obfuscate the relative distance between code objects, *e.g.* permutation-based defenses, can be derandomized by multiple disclosures. Fortunately, the use of runtime re-randomization to nullify leaked pointers [12, 31, 57, 58, 59, 186] and in-memory traps that throw security exceptions [190] have been instrumental at fortifying address obfuscation techniques. However, systems with these added protections suffer from far greater performance overheads (14%-225% for function permutation [12, 186]), reducing the appeal of randomization-based defenses.

The deficiencies of low entropy, memory disclosures, and high overheads have hindered the adoption of software-based address obfuscation techniques. Secure hardware integrated

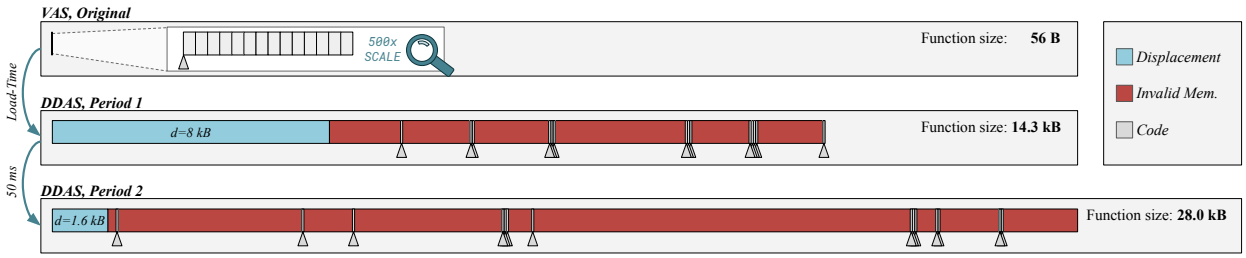


Figure 4.1: **DDAS Overview.** At load time, a 56 B function is displaced by 8 kB and dilated to 14.3 kB within the Displaced and Dilated Address Space (DDAS). Invalid memory regions are inserted between instructions and trigger security exceptions when accessed. Every 50 ms, the function layout re-randomizes, changing in both size and location. In DDAS, this function can be maximally inflated to 14 MB, and the entire address space can be dilated by over 250,000 times. DDAS addresses are derandomized before entering the virtual memory system, thereby eliminating performance impacts on the memory system.

into the processor pipeline can bridge this gap. Yet, existing hardware-based defenses, including the Morpheus secure architecture, are far behind their software counterparts, as they do not obfuscate relative distances [31]. In this work, we leverage custom RISC-V hardware to implement a defense with *all* of these necessities: high entropy, fine granularity, re-randomization, security exceptions, and low overheads. To protect code pointers from external influence, we completely decouple pointers stored in the data segment from the true location of their targets by expressing all code pointers in the superimposed ***Displaced and Dilated Address Space (DDAS)***. Within DDAS, we introduce address space ***displacement***, a low cost, one-time shift of the base of the code segment. To obfuscate the relative distance between code objects, we introduce address space ***dilation***, the insertion of undefined memory regions at an instruction-level granularity. Layered atop architectures like Morpheus, DDAS has the potential to stop attackers from learning the contents of the code segment (*i.e.*, code addresses or pointers) by introducing a high-entropy, frequently changing landscape.

Figure 4.1 illustrates the obfuscation of a function in DDAS. Since displacement occurs in the superimposed address space rather than virtual memory, the code segment can be relocated anywhere. Thus, we are able to shift the address space by a large, 64-bit key. To implement dilation, we leverage the vast unused portions of the address space. For example, in RISC-V, the upper sixteen and lower two bits of code pointers are unused, implying that we can dilate memory by 2^{18} times with no loss of usable space. Similar spans of unused addresses are available in the ARM and x86-64 instruction sets.

Lastly, we defend against memory disclosures by employing both in-memory traps and runtime re-randomization. Since we dilate memory programmatically, we can detect accesses

to the undefined memory regions that interleave instructions, effectively creating traps that detect attackers' attempts to leak code pointers. When these regions are accessed, the system triggers an ***immediate security exception***. Our analysis shows that, on average, 99.996% of DDAS consists of untouchable holes, and therefore attackers have less than a 0.01% likelihood of randomly guessing a valid code pointer (*i.e.*, a pointer to addressable memory) without detection. To protect against potential memory disclosures, we re-randomize the addresses of code objects at runtime, effectively hardening this technique against advanced probing and side-channel attacks that have the potential to derive unknown memory locations.

By implementing DDAS in hardware, our defense is transparent to programmers while providing high entropy protections with low overheads. DDAS is integrated in the processor pipeline via custom functional units for pointer translations and secure storage for keys that parameterize the memory layout. Additional modifications, including a second lightweight core, are used to implement re-randomization. We maintain the program counter and indexing of microarchitectural structures as hardware-derandomized, virtual address space (VAS) pointers, allowing us to completely eliminate potential performance impacts within the memory system. With hardware support, our defense has negligible performance overheads, at $\sim$1% with re-randomization every 50 ms, and it makes control-flow hijacking attacks impracticable to execute.

4.1.1 Chapter Contributions

With Displaced and Dilated Address Spaces, we boost uncertainty in code addresses to a point where it is incredibly difficult to *i)* Forge code pointers and *ii)* Manipulate code pointers using relative distance information, effectively thwarting all known forms of control-flow attacks in the control plane. In support of this argument, this chapter makes the following contributions:

- We introduce displacement to obfuscate absolute code locations by shifting the address space by a 64-bit key. We introduce dilation to create a super-inflated code plane and detect code pointer corruption, dilating consecutive instructions by a 100 kB untouchable hole, on average.
- We present a Displaced and Dilated Address Space (DDAS) implementation on RISC-V that can be maximally inflated to over 250,000 times the size of the existing virtual address space. We also propose DDAS-R (with re-randomization), where the memory layout is reconfigured under running programs to thwart memory disclosure vulnerabilities.

- We show that our proposed DDAS-R implementation (Table-Based, 2k-entries) introduces 55 bits of entropy for relative locations and 64 bits of entropy for absolute locations, and induces minimal overheads, with a 1.07% average slowdown at 50 ms re-randomization for the SPEC CPU2006 benchmarks.

This chapter is organized as follows. Section 4.2 presents our address dilation technique. Section 4.3 details the hardware implementation, while Section 4.4 provides support for runtime re-randomization. Section 4.5 presents our experimental setup, with Sections 4.6 and 4.7 analyzing the overheads (*i.e.*, performance, power, and silicon area) and security of our defense. This analysis is accompanied by a comparison to related work in Section 4.9. Section 4.10 concludes this chapter.

4.2 Obfuscating Code Locations with Displaced and Dilated Address Spaces (DDAS)

To defend against code reuse attacks in the control plane, we express all code pointers in a superimposed address space, termed the ***Displaced and Dilated Address Space (DDAS)***, layered atop the existing virtual address space (VAS). DDAS is addressable by 2^{64} distinct pointers, whereas the underlying VAS is only addressable by 2^{46} distinct pointers in RISC-V.[1] We leverage this fact to ***dilate*** the address space by over 250,000 times through the insertion of ***undefined memory locations*** in the DDAS space. These regions do not exist as accessible memory in the virtual address space, therefore any access to these holes results in an immediate security exception. Thus, to disclose or manipulate a code pointer, one must guess the value of the code pointer through high-entropy defenses without accidentally touching an undefined memory location. This approach makes it incredibly unlikely for an attacker to successfully disclose or manipulate a code pointer. Combined with runtime re-randomization (Section 4.4), this challenge is coupled with a countdown clock of typically 50 ms to exfiltrate a pointer and synthesize an attack.

In this section, we first overview our threat model for single-gadget, code reuse attacks on the control plane. Next, we present a simple, function-based DDAS implementation that dilates the address space uniformly. Then we examine a table-based DDAS implementation that trades simplicity to gain increased spatial diversity. We then present constraints on these implementations that allow for more efficient hardware implementations.

[1] Unused bits of pointers are also available in other architectures.

4.2.1 Security Goals

Our goal is to obstruct the acquisition of code pointer information assets to prevent the synthesis of control-flow attacks. Specifically, we aim to prevent the attacker from learning enough information about the victim's code layout to create a specific sequence of software inputs that, when provided to the victim application, without any assistance from hardware modifications (*e.g.*, Trojans), triggers a vulnerability to redirect program control flow. Specifically, we protect the following domains of implementation information:

- ***Code Pointers:*** The stored address of a code object, including function pointers, return addresses, and any addresses in memory

To achieve this goal, we randomize the absolute (I-2) and relative (I-3) location of code objects to obfuscate code pointers. We do not aim to prevent the creation of attack payloads that do not require knowledge of code pointer addresses. DDAS is applied to the control plane only. This defense could also be applied to data addresses in the data plane but may incur additional overheads.

4.2.2 Threat Model

The Displaced and Dilated Address Space (DDAS) is designed to protect against control-flow attacks that manipulate code pointers. Specifically, we focus only on control-plane attacks (Section 2.5.1)—although adapting DDAS to data pointers has the promise to also thwart data-plane attacks like DOP [64]. In our attack model, a powerful adversary is attempting to execute code gadgets, where the execution of a single gadget is indicative of a successful attack. The attacker can access a memory disclosure vulnerability to leak arbitrary information from the system and is able to analyze the program binary to derive the code layout. In addition, they have access to a memory corruption vulnerability that can be used to bootstrap gadget chains. While the attacker is knowledgeable about DDAS and the current re-randomization rate of the system, the keyed configuration of DDAS is secret at the start of the attack. Lastly, the system's random number source is assumed to produce true, non-repeating random numbers that cannot be predicted by the attacker.

Non-Goals. We do not thwart data-plane attacks or attacks that rely on the manipulation of data or data pointers (*e.g.*, DOP [64]). We do not prevent the injection of code in the data segment of the program (as done in a code injection attack). We assume existing protections like W⊕X [51] are in place to prevent this behavior.

4.2.3 Displacing and Dilating the Code Segment

Below, we present two dilatation schemes for the code segment: basic DDAS, which dilates the address space uniformly, and table-based DDAS, which dilates the address space via a repeating pattern of holes.

4.2.3.1 Basic DDAS Translation

The DDAS memory configuration is determined programmatically and all code pointers in memory are expressed as DDAS pointers. Consider a simplified DDAS memory layout that lends itself to efficient dilation and compression, as shown in Figure 4.2. Here, the address space is shifted by a single displacement and divided into constant sized segments that are dilated uniformly by an undefined memory region (*i.e.*, a hole), shown in red. The layout of DDAS is parameterized by three keys: the displacement size, d, the segment size, S_{vas}, and the dilation amount per segment, i.

In DDAS, all code pointers are represented as 64-bit DDAS values. Memory remains unmodified (*i.e.*, in VAS), thus code pointers must be translated from DDAS (A_{ddas}) to VAS (A_{vas}) addresses before use to maintain correctness. This transformation is conceptually similar to converting pointers from DAS to VAS in Morpheus (Section 3.2.3). Additionally, a check must be performed to assert that the pointer does not access an undefined memory region. Likewise, whenever a code pointer is created, it must reflect the DDAS value. These operations are performed with the following functions:

VAS→DDAS Translation:

$$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor * i) \tag{4.1}$$

DDAS→VAS Translation:

$$A_{vas} = A_{ddas} - d - (\left\lfloor \frac{A_{ddas} - d}{S_{ddas}} \right\rfloor * i) \tag{4.2}$$

Valid Memory Location Assertion:

$$(A_{ddas} - d) \bmod S_{ddas} \ < \ S_{vas} \tag{4.3}$$

Using these equations, we can translate between the VAS and DDAS segments and verify code pointers. If the above assertion ever fails, this indicates that a code pointer has been manipulated and a security exception is thrown.

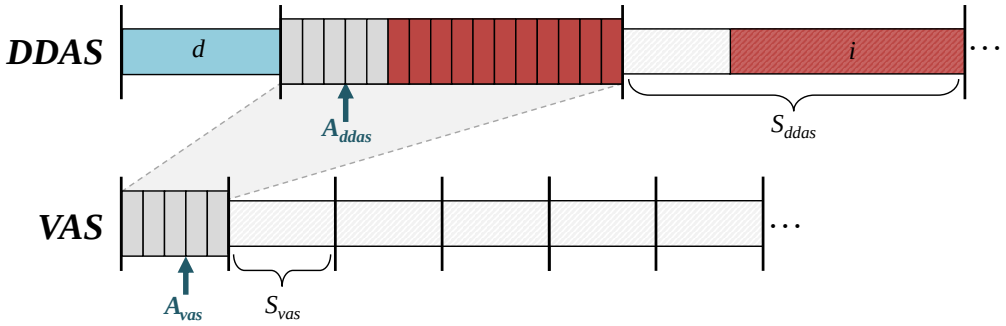


Figure 4.2: **Basic DDAS Design.** The VAS is divided into segments of size S_{vas}. Each segment is mapped into DDAS and dilated by an i byte hole. The address space is shifted by d bytes.

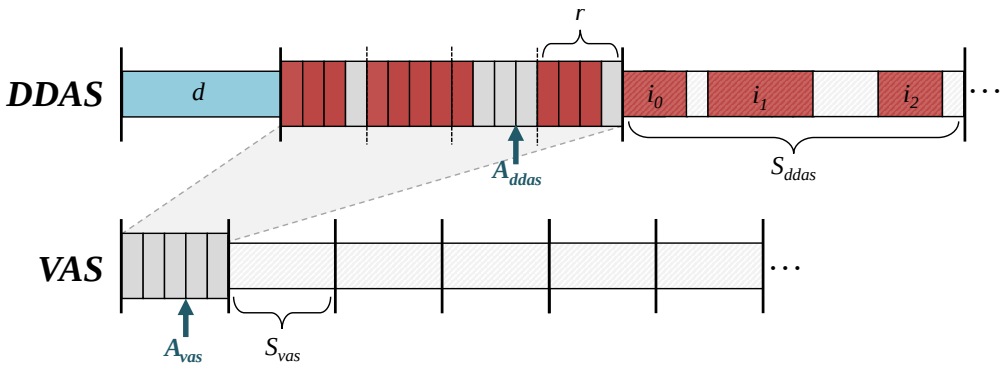


Figure 4.3: **Table-Based DDAS Design.** Varying sized holes (i_0-i_n) are inserted between the valid addresses in a single VAS segment. These holes sum to i bytes. Additionally, each segment is divided into smaller ranges of size r to efficiently track the locations of holes in the Range-Map table.

4.2.3.2 Table-Based DDAS Translation

Although basic DDAS is effective at inflating the address space, there is little spatial diversity as i is a constant, single hole. To increase the entropy of our defense, we present a table-based memory configuration, shown in Figure 4.3, where each segment is recursively divided into a *range* of size r. The i byte inflation is now distributed throughout these ranges as multiple, distinct holes of varying sizes. This increases diversity by varying the size and location of holes across the code space. The configuration of the holes within a segment is recorded by the *Range-Map table*. To translate from DDAS→VAS, the address's DDAS segment offset is used to index the table, T, returning the size of the holes within the segment that precede the queried address. This lookup also returns the size of the hole in the address's range, denoted T_r, which is used to validate the DDAS code pointer. In our implementation, we instead index T and T_r with the address's range to minimize the total number of table entries

required. To translate from VAS→DDAS, the address's VAS segment offset is used to index a separate, analogous table, T'.

Under this modified implementation, the layout of DDAS is parameterized by the Range-Map table, as well as the three keys used previously, d, i, and S_{vas}, and a new key, r. The operations for translating code pointers between DDAS addresses (A_{ddas}) and VAS addresses (A_{vas}) and asserting that code pointers are valid are given by the following functions:

VAS→DDAS Translation:

$$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor * i) + T'[A_{vas} \bmod S_{vas}] \tag{4.4}$$

DDAS→VAS Translation:

$$A_{vas} = A_{ddas} - d - (\left\lfloor \frac{A_{ddas} - d}{S_{ddas}} \right\rfloor * i) - T[R_{ddas}] \tag{4.5}$$

Where the DDAS range number (R_{ddas}) is calculated via:

$$R_{ddas} = \lfloor ((A_{ddas} - d) \bmod S_{ddas})/r \rfloor \tag{4.6}$$

Valid Memory Location Assertion:

$$(A_{ddas} - d) \bmod S_{ddas} \bmod r \ \geq \ T_r[R_{ddas}] \tag{4.7}$$

With this implementation, we can support dilation between *every instruction* in the code segment. Each table entry must correspond to a single 4 B instruction in the virtual address space to dilate at an instruction-level granularity. Since the table is implemented in hardware and its size is fixed, this constraint implies that S_{vas} can be no more than four times the number of table entries. We analyze the performance-security trade-offs of different table sizes in Section 4.7 and Section 4.6.

4.2.4 Architecting an Efficient DDAS Implementation

The keys that parameterize DDAS can be constrained to match the desired performance-security trade-offs of the application. Here, we focus on optimizing the DDAS→VAS translation and valid memory location assertion as these two functions must be computed on every indirect jump. By constraining S_{ddas} to be a power of two, we can replace the division operations in Equations 4.2 and 4.5 with a right shift of $\log_2(S_{ddas})$ bits, and the modulo

Basic DDAS	***Constraints:*** i and S_{ddas} are a power of 2	
VAS→DDAS	$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor * i)$	Eq. 4.1
	$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor \ll \log_2(i))$	Eq. 4.1-c
DDAS→VAS	$A_{vas} = A_{ddas} - d - (\left\lfloor \frac{A_{ddas}-d}{S_{ddas}} \right\rfloor * i)$	Eq. 4.2
	$A_{vas} = A_{ddas} - d - (((A_{ddas} - d) \gg \log_2(S_{ddas})) \ll \log_2(i))$	Eq. 4.2-c
Valid Address Assertion	$A_{ddas} - d \ < \ (\left\lfloor \frac{A_{ddas}-d}{S_{ddas}} \right\rfloor * S_{ddas}) + S_{vas}$	Eq. 4.3
	$A_{ddas} - d \ < \ (((A_{ddas} - d) \gg \log_2(S_{ddas})) \ll \log_2(S_{ddas})) + S_{vas}$	Eq. 4.3-c
Table-Based DDAS	***Constraints:*** r and S_{ddas} are a power of 2	
VAS→DDAS	$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor * i) + T[A_{vas} \mod S_{vas}]$	Eq. 4.4
	$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor * i) + T[A_{vas} \mod S_{vas}]$	Eq. 4.4-c
DDAS→VAS	$A_{vas} = A_{ddas} - d - (\left\lfloor \frac{A_{ddas}-d}{S_{ddas}} \right\rfloor * i)$ $-T_a[((A_{ddas} - d) \mod S_{ddas})/r]$	Eq. 4.5
	$A_{vas} = A_{ddas} - d - (((A_{ddas} - d) \gg \log_2(S_{ddas})) * i)$ $-T_a[((A_{ddas} - d) \ \& \ (\text{0xffffffff} \gg \log_2(S_{ddas}))) \gg \log_2(r)]$	Eq. 4.5-c
Valid Address Assertion	$((A_{ddas} - d) \mod S_{ddas}) \mod r$ $< \ T_c[((A_{ddas} - d) \mod S_{ddas})/r]$	Eq. 4.7
	$((A_{ddas} - d) \ \& \ (\text{0xffffffff} \gg \log_2(S_{ddas}))) \ \& \ (\text{0xffffffff} \gg \log_2(r))$ $< \ T_c[((A_{ddas} - d) \ \& \ (\text{0xffffffff} \gg \log_2(S_{ddas}))) \gg \log_2(r)]$	Eq. 4.7-c

Table 4.1: **Summary of DDAS Translation Equations.** This table summarizes the equations needed to implement basic or table-based DDAS. As detailed in Section 4.2.4, multiply, divide, and modulo operations are simplified by constraining the key parameters. The optimized/constrained equations are indicated via the notation "-c".

operators in Equations 4.3, 4.6, and 4.7 with a bitmask that selects the upper $64-\log_2(S_{ddas})$ bits. These values are stored locally for efficient computation.

Constraining S_{ddas} removes lengthy operations on the critical path. Furthermore, by constraining other key parameters, we can further simplify outstanding multiply, divide, and modulo operations. For example, r must be a power of two since it is a divisor of S_{ddas}. This constraint further simplifies Equations 4.6 and 4.7. We can similarly constrain the hole size, i, to be a power of two to eliminate the multiplication in Equations 4.1, 4.2, 4.4, and 4.5. We employ this optimization when a multiply instruction is on the critical path, which occurs only for the Basic DDAS configuration. These optimizations are summarized in Table 4.1.

4.3 The DDAS Architecture

We implement DDAS on a modified RISC-V architecture [192]. The baseline architecture is extended with functional units to translate code pointers between the DDAS and VAS address spaces as necessitated during runtime. These extensions are shown in Figure 4.6.

DDAS only impacts indirect jumps during program execution. Thus, the pipeline only needs to be instrumented with a small, dedicated functional unit to translate code pointers before use. These custom hardware extensions enable a more efficient implementation compared to software instrumentation. Instrumenting binaries to perform pointer translations before indirect control transfers would be costly, even when indirect jumps make up a small percent of instructions (about 2.4% in our analyzed workloads). Hardware support not only reduces performance impacts, but also enables us to achieve a strong layer of isolation between the software and the secret keys that parameterize DDAS, as these keys are stored in dedicated hardware registers guaranteed to remain secret. Below, we present the hardware extensions to support load-time DDAS pointer randomization. In Section 4.4, we discuss extensions that support runtime re-randomization of DDAS pointers, similar to Morpheus churn (Section 3.3.2).

4.3.1 Hardware Architecture

The use of unaligned, 64-bit DDAS code pointers introduces a layer of indirection that requires pipeline modifications to ensure correct control flow. To reduce the impact on direct jumps and instruction fetch, the program counter (PC) is maintained as a VAS value. Thus, no modifications to these operations that directly use the PC are required. Further details on the modified representation of code pointers' impact on program execution are below.

Direct Jumps (JAL/BR): Direct jumps add a constant offset to the program counter, encoded in an immediate. Since we maintain the program counter in VAS, no modifications are made for direct jumps. To our knowledge, this does not have security implications as direct jump targets are encoded in the instruction and cannot be modified by malicious inputs.

Indirect Jumps (JALR): Indirect jumps handle function calls and returns, as well as other arbitrary control flow, by loading a code pointer from a register into the program counter. With DDAS, all code pointers stored in registers are DDAS values. Therefore, before loading the register value into the program counter, the pointer must be asserted as

valid and translated to its VAS equivalent. This translation occurs implicitly in the microarchitecture through a ***dedicated functional unit***, detailed in Section 4.3.2. Additionally, when the link register is written, the microarchitecture translates the $(pc + 4)$ link address from a VAS to a DDAS address.

Code Pointer Provenance (LUI/AUIPC): The targets of function calls are formed at runtime via a `lui` (or `auipc`) and `addi` instruction sequence. To generate a DDAS pointer rather than a VAS pointer, the compiler relocates code pointers to the global segment, which are then translated to DDAS values by the program loader. This implementation allows us to configure DDAS pointers without injecting backdoor instructions that perform VAS→DDAS translations at runtime.

Code Pointer Arithmetic (ADD, ADDI, *etc.*): DDAS is designed to thwart attacks that use relative distance information and arithmetic to forge code pointers. While DDAS does not prohibit code pointer arithmetic, it randomizes the result. Using a modified code pointer will result in an access to an unintended valid memory location or, with high probability, to an invalid hole. Thus, code pointer arithmetic is not advised in DDAS. We argue that arithmetic on code pointers is not normal program behavior, as the C1x Standard [193] does not allow arithmetic operations on pointers to function types or incomplete types, like `void*` (§6.5.6-2 & §6.2.5.1). Although the GNU extension for C does permit operations on code pointers, DDAS supports this at the program's peril.

4.3.2 DDAS Unit Microarchitecture

To maintain correctness, all indirect control-flow transfers are preceded by a DDAS→VAS translation and valid pointer assertion. We instrument the execute stage of the pipeline with a dedicated functional unit to perform this computation for each indirect jump target (A_{ddas}). The latency of this translation is masked by the writeback stage, which is idle for indirect jump instructions. Figures 4.4 and 4.5 show the functional units for basic DDAS and table-based DDAS, respectively. The Range-Map table is implemented as a fixed-size memory structure with 8 B entries. We evaluate the exposed latency of these functional units with different table sizes in Section 4.5.

4.3.3 Key Generation

The DDAS memory configuration is determined by secret parameters generated at load-time. For basic DDAS, a true random number generator (TRNG) is queried to determine the three

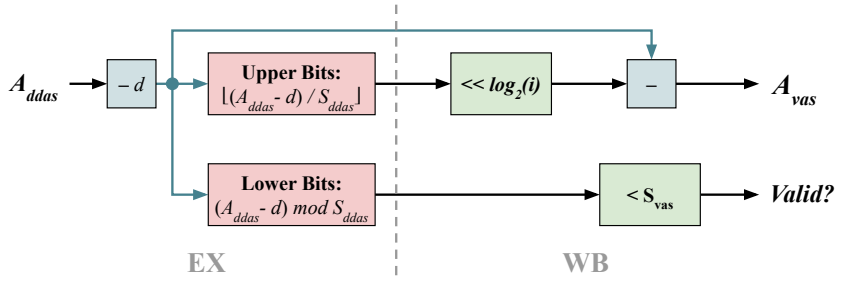


Figure 4.4: **Functional Unit for Basic DDAS.** This unit computes Equations 4.2 and 4.3 in parallel for a given code pointer, A_{ddas}.

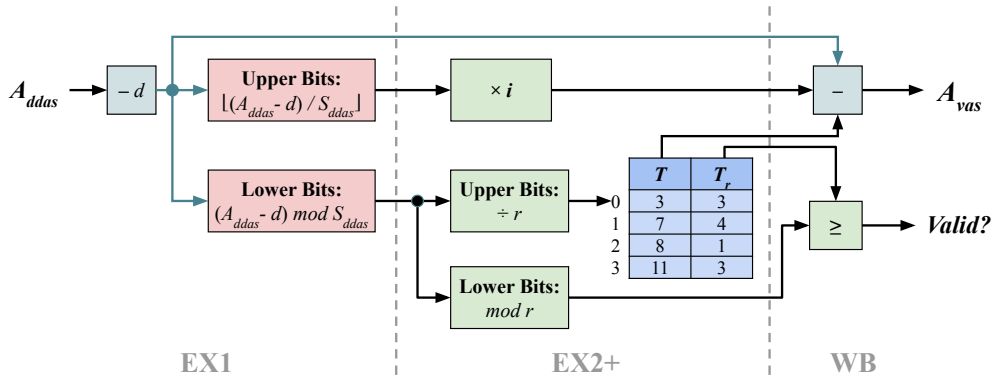


Figure 4.5: **Functional Unit for Table-Based DDAS.** This hardware unit computes Equations 4.5-c and 4.7-c. The Range-Map table, indexed by the range number, returns T for the A_{vas} calculation and T_r for the valid pointer assertion.

keys: d, i, and S_{vas}. These values are stored in dedicated hardware registers. For table-based DDAS, r and a Range-Map key are generated in addition to those above. The Range-Map key is used to seed a pseudorandom number generator (PRNG) to deterministically populate the Range-Map table. This optimization enables efficient context switching, as described below.

4.3.4 Context Switching Support

To be a viable defense in modern systems, the DDAS hardware must support context switching between individual processes that have unique DDAS configurations (*i.e.*, separate keys). During a context switch for basic DDAS, its keys are encrypted under a boot-time hardware key before being written out to memory. Context switching for the table-based implementation requires storing the range size, r, as well as the Range-Map key, which is used to seed the PRNG to generate the Range-Map table. When a process switches back in, the offloaded DDAS parameters are loaded and decrypted to be used again. For the table-based configuration, the Range-Map key seeds the PRNG and generates the same table layout as before. With this optimization, the contents of the Range-Map table do not need to be encrypted

and written to memory, saving a considerable amount of time given our large table sizes.

4.4 Adding Runtime Re-Randomization (DDAS-R)

DDAS obfuscates the relative and absolute locations of code objects at runtime. However, attacks may attempt to guess or probe the system for code locations to subvert the defense. The research community has frequently cited re-randomization as a protection mechanism for both derandomization probes and memory disclosures [12, 31, 57, 58, 59, 186]. Additionally, re-randomization is effective at deterring multi-stage attacks that require the discovery and use of numerous code gadgets. DDAS with runtime re-randomization (DDAS-R) implements churn by periodically re-keying the memory configuration and updating all code pointers accordingly. This creates a hostile address space that inflates and contracts, containing moving undefined memory locations that must be avoided.

When a re-randomization cycle begins, the pipeline is flushed to resolve in-flight instructions that contain stale code pointers. Then, custom hardware generates new keys to parametrize the re-randomized DDAS memory layout. These keys are loaded into dedicated hardware structures and the code pointers in registers are updated to reflect the new keys. At this point, program execution is resumed. All code pointers in the caches and DRAM are currently stale, *i.e.*, use the old key set. In parallel with program execution, these values are updated by the DDAS Remapper—a small, finite state machine that is responsible for pointer updates. The Remapper uses a DDAS→VAS and subsequent VAS→DDAS functional unit to translate a pointer from the old to new key set.[2]

A new re-randomization cycle can commence as soon as the memory scan in the prior period terminates. In our analysis, most benchmarks take no more than a few milliseconds to re-randomize, depending on their memory footprint. However, operating at high re-randomization rates can incur increased performance overheads (Section 4.6). In this work, we recommend a 50 ms re-randomization period because it is sufficiently faster than existing exploits without introducing much overhead. Namely, JIT-ROP exploits have been recorded to take 2.3 seconds or more [23], whereas Blind-ROP takes a minute on `nginx` [24]. Additionally, other interval-based re-randomization defenses (*e.g.*, Shuffler [12] and Morpheus [31]) operate at 50 ms with this justification.

[2]This process is very similar to Morpheus churn (Section 3.3.2).

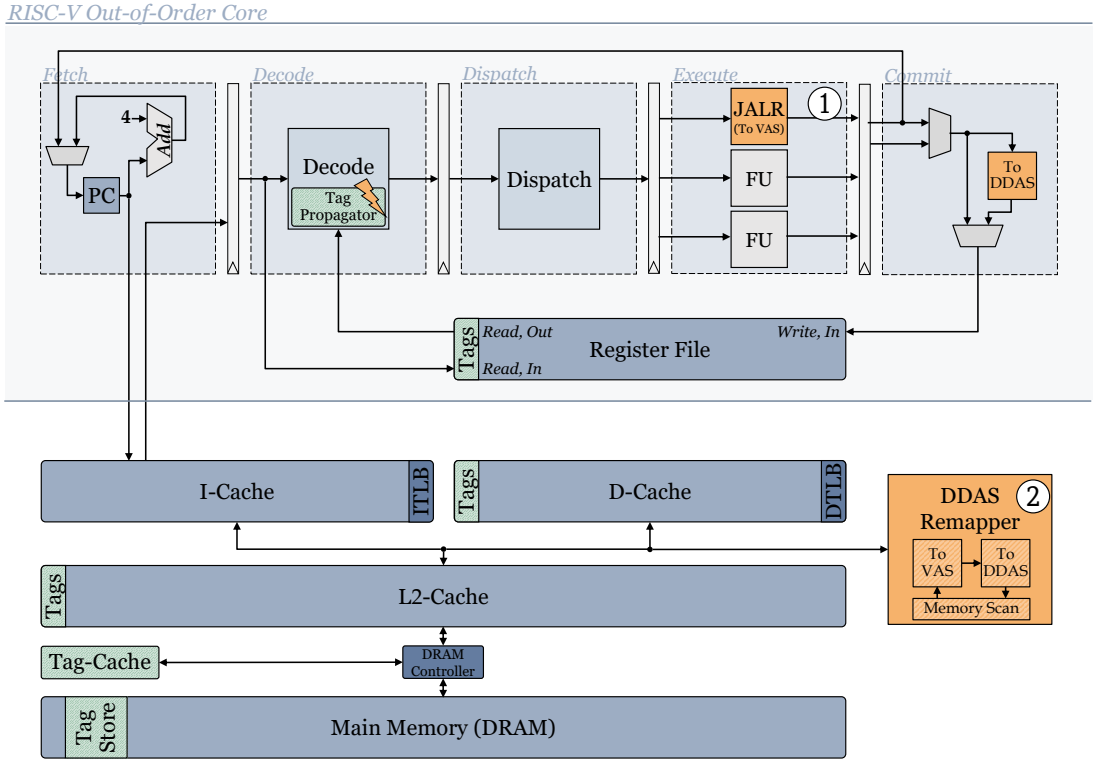


Figure 4.6: **DDAS(-R) Architecture.** The baseline RISC-V architecture is extended with a DDAS→VAS functional unit ① to translate code pointers during `JALR` instructions. The microarchitecture of these functional units are shown in Figures 4.4 and 4.5 (above). In DDAS-R, these functional units are extended to accommodate the mixed state of pointers during re-randomization. The DDAS Remapper ② uses tagged memory to identify and update stale code pointers.

4.4.1 Hardware Support

Hardware extensions to support re-randomization are shown in Figure 4.6. Similar to prior work [31], tagged memory is used to locate code pointers and identify the stale code pointers that require updating. The L1- and L2-caches are extended by a 1-bit tag for every pointer-width word (8 bytes). In physical DRAM, these tags are stored in a fixed block determined statically by the physical address. Tag information is obtained by an LLVM IR pass and loaded into the system via binary metadata. Lastly, the pipeline is extended to support the propagation of tag information throughout the system.

The processor is augmented to include the DDAS Remapper, a small hardware unit responsible for key generation and code pointer updates. When a periodic re-randomization cycle begins, the DDAS Remapper generates a new set of DDAS keys and loads them into the associated structures before program execution resumes. The remapper then scans tag information sequentially to identify stale code pointer values, which are loaded into the remapper and updated using internal functional units. The new pointers are written back

via a cache-coherent bus between the L1 and L2. The current VAS address being processed, termed the *threshold* value, is stored to indicate the progress of the remapper. All memory values above this address have been updated with the new key.

Maintaining Correctness for Mixed State Since both updated and stale code pointers exist in memory during re-randomization, the pipeline must accommodate this mixed state. The functional unit is expanded to store both the new and old key sets and, for table-based DDAS, the block size of each Range-Map entry is doubled from 8 B to 16 B. During a load, code pointers are compared to the threshold value to determine if they are using the new or old key set. This result is reflected in a 1-bit register tag, which is fed into the functional unit to select the correct key set and table entry, maintaining correctness. Lastly, when a stale code pointer is stored to a region of memory that has since been updated, it must be updated with the new key set before being written back. Our simulations show that this case is rare and has negligible performance impacts.

4.4.2 Support for Context Switching in DDAS-R

When the DDAS Remapper is idle during a context switch (*i.e.*, all code pointers are updated), support for context switching is equivalent to DDAS without re-randomization (Section 4.3.2). However, when a context switch occurs during a re-randomization cycle where the remapper is active, the complete DDAS context must be encrypted and stored to memory. This information includes both the new and old sets of DDAS keys and the threshold value, which indicates the progress of memory updates in the current re-randomization cycle. When a context is switched-in, the threshold value is loaded into the DDAS Remapper and re-randomization begins from this virtual address. Since a process may be switched-out for longer than the specified re-randomization period (*e.g.*, 50 ms), the remapper initiates another re-randomization cycle immediately upon completion to ensure that any code pointer's lifetime in an old key set is as short as possible.

4.5 Experimental Methodology

We implemented DDAS on a RISC-V out-of-order core in gem5 [6, 148] in system call emulation mode with the parameters listed in Table 4.2. We modeled the memory contention for tagged memory and pointer updates using DRAMSim2 [149]. To determine the latency of the DDAS functional unit, we modeled the Range-Map table in CACTI 7 [9] at the 28 nm technology node. To analyze the security and performance trade-offs of our defense, we

examined three distinct DDAS configurations with load-time and runtime randomization– *i)* Basic DDAS where i is restricted to a power of two (1 cycle of exposed latency), *ii)* Table-based DDAS with a 2k-entry table (2 cycles of exposed latency), and *iii)* Table-based DDAS with a 32k-entry table (4 cycles of exposed latency).

Core Type	2.5GHz, O3CPU (Out-of-Order)
Superscalar	4-wide
Cache Line Size	64B
L1 Cache	32KB with 2-cycle latency
L2 Unified Cache	256KB with 20-cycle latency

Table 4.2: **Experimental Configuration for gem5.** The specifications of our baseline RISC-V out-of-order core in gem5 [6].

Compiler Support and Benchmarks. We implement our compiler passes in LLVM 5.0.0 configured for the RISC-V RV64IMA architecture [153]. All programs were compiled with optimization level -O2 and linked against the RISC-V Musl C library [154]. Function pointers that are normally formed by arithmetic code sequences during runtime are relocated to the global segment so they can easily be randomized by the gem5 loader or by the DDAS Remapper. We have found that this simple compiler change on average speeds up programs by ∼1.85%. We also analyzed the size increase of the binaries and found that they only increase by 1.8% on average. Since one could migrate code pointers to the static global section, we lend this optimization to the baseline in order to reveal the performance overheads incurred by running programs due to the DDAS hardware. We evaluate DDAS on the first three billion instructions of the SPEC CPU2006 C-code benchmarks [151, 152] with the reference input: *perlbench*, *bzip2*, *gcc*, *mcf*, *milc*, *gobmk*, *hmmer*, *sjeng*, *libquantum*, *h264ref*, *lbm*, and *sphinx3*.

4.6 Experimental Results and Evaluation

To measure the performance impacts of DDAS, we ran experiments against the SPEC CPU2006 C workloads [151, 152], described in Section 4.5, for the chosen configurations– basic (1 cycle latency), table-based, 2k-entry (2 cycles latency), and table-based, 32k-entry (4 cycles latency).

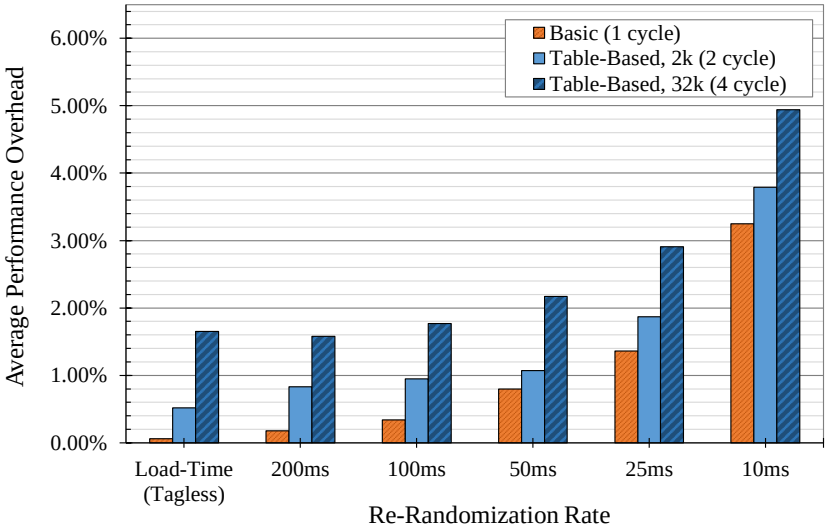


Figure 4.7: **Average Performance Overheads.** The average overhead for the SPEC CPU2006 benchmark suite at varied re-randomization rates for the analyzed DDAS configurations.

4.6.1 Runtime Performance Overheads

Figure 4.7 shows the average performance overhead of these configurations for the analyzed workloads. The DDAS functional unit introduces latency in the pipeline that can delay branch prediction and cause bottlenecks. The overhead for a load-time (tagless) DDAS system illustrates the consequences of this added latency alone. To support re-randomization, tags for every pointer-width word must be stored and propagated in the memory hierarchy. Additionally, the re-randomization process accesses tag information and rewrites all code pointer values in the data segment. These updates invalidate code pointers in the data cache, increasing miss rates. Thus, compared to the load-time system, re-randomization incurs higher performance penalties and affects the memory system.

Our recommended configuration, DDAS-R with a 2k Range-Map, has reasonable slowdowns, at 1.07% during 50 ms re-randomization. A detailed breakdown of the overhead incurred by individual benchmarks for this configuration is shown in Figure 4.8. Performance overheads generally increase alongside re-randomization rates, especially for benchmarks with a large number of code pointers, *i.e. gcc* has 5.18% slowdowns at 50 ms. For these benchmarks, the DDAS Remapper must perform more pointer updates, invalidating values in the L1-cache and increasing both the number of read/write misses and the overall average miss latency for the main core. Some benchmarks with more `jalr` instructions experienced reduced slowdowns at faster re-randomization rates. This is likely a result of the DDAS remapper prefetching code pointers into the caches during re-randomization, effectively making these values more available to the core. However, as re-randomization rates

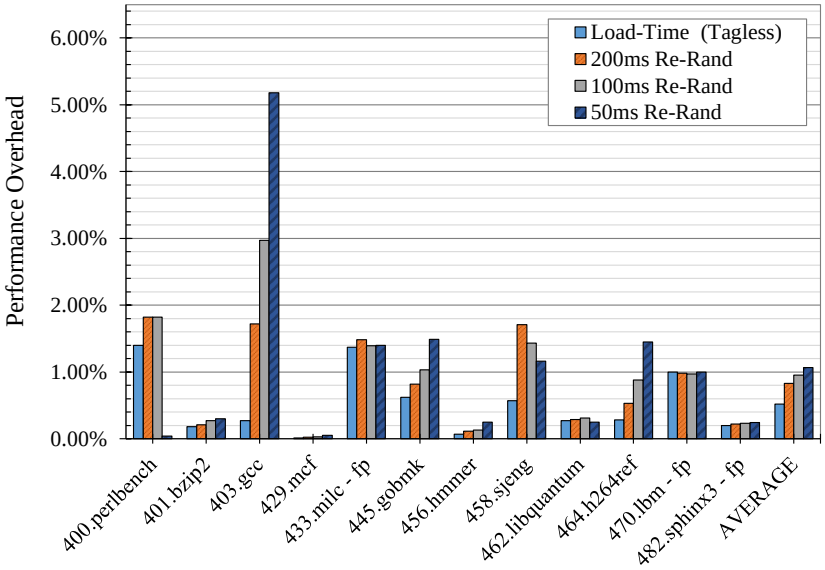


Figure 4.8: **Performance Overheads for Table-Based, 2k DDAS.** The performance overheads per benchmark for a table-based, 2k-entry DDAS configuration, with 2 cycles of exposed latency.

increase, this benefit will likely diminish due to interference during pointer updates.

4.6.2 Power and Area Overheads

The hardware overheads of DDAS were estimated using McPAT 1.3 at the 28 nm technology node [194]. We used their ARM Cortex-A9 out-of-order processor as a template for our RISC-V system, which was augmented with: *i)* The DDAS functional unit, *ii)* Memory system modifications (*i.e.*, tagged memory and the Range-Map table) modeled in CACTI 7 [9], and *iii)* The DDAS Remapper, estimated by SiFive's S51 64-bit in-order core [8]. Table 4.3 shows the estimated area and power consumption for the analyzed DDAS configurations. The cost to implement table-based DDAS with 2k entries is fairly low, at less than 3% area overhead for a system without load-time randomization. With re-randomization, this overhead increases to about 7%. For a table-based configuration with 32k entries and re-randomization support, the area overhead is much higher at nearly 40%. Unfortunately, for large tables, support for re-randomization becomes expensive as table entries are doubled in size to accommodate both key sets.

4.7 Security Analysis

DDAS prevents code reuse attacks on the control plane by obfuscating the absolute and relative addresses of code objects. Unlike prior approaches, which permute code objects or encrypt pointers, DDAS does more than obfuscate the code segment. DDAS also catches

	Area (mm^2)	Overhead	Power (W)	Overhead
Baseline	4.387	-	3.653	-
Basic DDAS	4.411	0.55%	3.674	0.57%
Basic DDAS-R	4.524	3.13%	3.682	0.80%
Table-Based DDAS, 2k	4.506	2.71%	3.704	1.40%
Table-Based DDAS-R, 2k	4.690	6.90%	3.804	4.13%
Table-Based DDAS, 32k	4.865	10.88%	3.850	5.39%
Table-Based DDAS-R, 32k	6.111	39.29%	4.370	19.61%

Table 4.3: **Hardware Estimation.** The area and power estimates for the analyzed DDAS configurations.

when a pointer is overwritten by detecting any access to the undefined regions of memory that make up a majority of the address space. Notably, in table-based, 2k-entry DDAS, on average more than 99.996% of the address space is undefined memory. Of the remaining 0.004%, only a fraction is mapped and in the code segment. Thus, there is a high likelihood that any attempt to forge a code pointer will result in an exception from either accessing an invalid memory region, an unmapped region of memory, or the data segment. Additionally, frequent re-randomization of the memory layout during runtime prevents the reuse of leaked code pointers, effectively thwarting control-plane attacks.

Below we qualitatively reason about why DDAS prevents sophisticated code reuse attacks, including return-oriented programming (ROP) [55] and its variants, as well as Spectre Variant 2. Since DDAS is a probabilistic defense, the subsequent sections quantify the entropy of DDAS to demonstrate its effectiveness at stopping these control-plane exploits.

4.7.1 Qualitative Discussion of Security Against Common Attacks

ROP. To dispatch ROP gadgets, a memory corruption vulnerability is exploited to overwrite a return address with a code pointer. In DDAS, this pointer is unknown and randomized every 50 ms on average. Additionally, if DDAS is maximally inflated, the attacker has less than an 0.001% chance of generating a valid code pointer, *i.e.* one to addressable memory. Variants of ROP that exploit other control-flow statements, like jump-oriented programming [67], also fall under this category.

JIT-ROP. JIT-ROP [23] uses a leaked code pointer to read code pages and construct gadgets at runtime. This leaked pointer is in the form of a DDAS address, so it cannot be used alone to derandomize the address space. Furthermore, a load using a DDAS pointer will not be translated to a VAS value, resulting in an incorrect effective address. Even if

	Average Inflation per Insn	Average Inflation per Page	Average Percent of Holes	Relative Entropy
Basic	78.9 kB	80,650 kB	99.994%	54.89 bits
Table-Based, 2k	107.8 kB	111,341 kB	99.996%	55.30 bits
Table-Based, 32k	111.3 kB	111,784 kB	99.996%	55.31 bits

Table 4.4: **Evaluation of DDAS.** The average inflation per instruction or page is the mean dilation incurred by a jump of that length in VAS. The table also includes the average percentage of memory occupied by untouchable holes and the calculated relative entropy (Section 4.7.2.2) per configuration.

an attacker can leak enough information to create a gadget chain, they must complete their attack before the next re-randomization cycle as all pointers will become stale.

Blind-ROP. Blind-ROP [24] repeatedly overwrites a single byte of the return address in a child process until the process does not crash, signaling that the guess was correct. This process is repeated for each subsequent byte to recover the entire pointer. With DDAS, these guesses will trigger exceptions with high probability. Additionally, this attack requires on average 1,500 requests to discover the ASLR offset and was recorded to have taken 1 minute on *nginx*. We randomize 1,200 times faster than the length of this attack at a period of 50 ms.

Spectre Variant 2. In Spectre Variant 2 [16], an attacker process mimics the control flow of a victim process to train the branch predictor to mis-speculate on a chosen indirect jump. This causes the victim process to execute an attacker-specified code gadget. In this exploit, an attacker must know the address of the target indirect jump instruction in the victim program. In DDAS, the attacker and victim process has different shifted and dilated code segments, preventing attackers from being able to understand and mimic the victim's jump patterns.

4.7.2 Quantitative Security Analysis of DDAS Entropy

Since DDAS provides probabilistic security, we quantitatively analyze the strength of our defense for our three studied configurations. Specifically, we analyze the likelihood an attacker can forge a valid code pointer at runtime. In our analysis, we run a 100,000 trial Monte Carlo simulation, where each trial has a randomly generated DDAS key set. For each trial, we record the amount of inflation incurred by an indirect jump in the VAS segment, starting from a jump to the next instruction. Additionally, for each indirect jump distance,

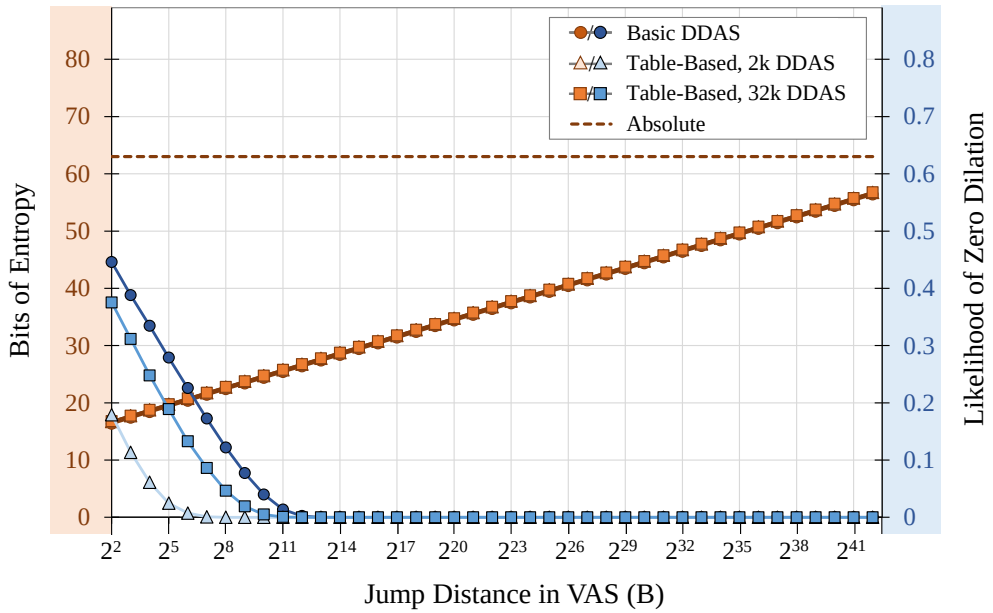


Figure 4.9: **Entropy of Indirect Jumps.** The primary axis, in orange, shows the entropy of indirect jumps, where entropy is modeled as log_2 of the dilation in bytes. The secondary axis, in blue, shows the probability that a given key configuration will *not* impact the distance of an indirect jump.

we measure the inflation from 100 randomized starting locations per trial. We then use these measurements to derive statistics about the overall inflation of the control plane, including the likelihood that a specific jump will encounter no dilation.

4.7.2.1 Inflation of the Control Plane

Our results, summarized in Table 4.4, show that table-based DDAS inflates the address space more than basic DDAS, dilating a 4 B jump by 107.8 kB or 111.3 kB on average, compared to 78.9 kB. The standard deviation of these metrics is also telling. For a 4 byte jump, the standard deviation is 2086 kB, implying that there is a large variance in the inflation of the code segment, even for very short jumps. Page-length jumps were inflated by over 111,000 kB on average with a standard deviation of 200 MB. These results suggest that a malicious code gadget cannot be more than a few bytes away. Otherwise, the high entropy of dilation forces attackers to resort to brute-force guessing. Furthermore, not only does guessing a pointer value subject an attacker to accessing unmapped memory, as in ASLR [5], but it also subjects them to accessing an untouchable hole and triggering an immediate security exception. Our analysis shows that a large portion of the address space is filled with these untouchable holes. Namely, in basic DDAS, 99.994% of memory on average contains untouchable holes, compared to 99.996% for both table-based configurations.

4.7.2.2 Entropy of Indirect Jumps

To measure the entropy of our defense, we consider the obfuscation of both absolute and relative addressing in the control plane. We define ***absolute entropy*** as log_2 of the average bytes of displacement. The uncertainty of the absolute location of code objects is attributed to the displacement key, d, which is 64 bits in length, giving a mean displacement of 2^{63} bytes. Thus, the absolute entropy is 63 bits. Similarly, we define ***relative entropy*** as log_2 of the mean bytes of dilation, averaged across varying jump distances. Previously, we quantified the entropy of dilation for specific jump distances, as shown on the primary axis of Figure 4.9. To measure the relative entropy of our whole defense, we repeat this calculation across *all* jump distances and find that our defense has an average relative entropy of 55 bits. This metric captures the overall difficulty of guessing the dilation for a jump of *any* distance. We find that the relative entropy of DDAS is well above existing defenses, like Shuffler [12], which typically has no more than 27 bits of entropy.

4.7.2.3 Likelihood of Zero Dilation

One limitation of DDAS is that, when a region of memory experiences zero inflation, the manipulation of jump targets within that range will succeed. This prompts us to evaluate how often zero inflation occurs for randomly chosen key configurations. We performed this analysis using the above simulation, now recording the frequency of zero dilation. The result of this analysis is shown in Figure 4.9. Basic DDAS always inflates jumps greater than 8 kB. Table-based DDAS performs better, inflating all jumps greater than 4 kB (a page). For jumps within a page, there is a small likelihood that no dilation occurs. Table-based DDAS with 2k entries achieves a 95% likelihood of dilation for jumps greater than 32 B. Table-based DDAS with 32k entries and basic DDAS only achieves this likelihood for jumps greater than 256 B and 1024 B, respectively. For comparison, a function in our benchmarks is 890 B on average. Thus, limited dilation for short relative distances can be tolerated and would restrict attackers to using code gadgets within the current function. Additionally, with re-randomization, these uninflated jumps would not persist for long (50 ms on average).

Overall, the DDAS configurations evaluated have comparable entropy, but varying amounts of inflation for short jumps. Notably, although table-based, 2k-entry DDAS has a smaller keyspace (S_{vas} is constrained by the number of table entries), it is more effective than 32k-entry DDAS at inflating short jump distances. This is because, as the VAS segment becomes larger, the possibility for consecutive valid memory locations increases, causing the probability of dilation for short jumps to decrease. Given these results, ***we recommend table-based DDAS with 2k entries***. We find the performance overheads for

this configuration to be reasonable, as presented in Section 4.6.

4.7.3 Attacking DDAS

With the DDAS secure hardware, existing control-plane exploits will be unsuccessful due to the obfuscation of code pointers in a high-entropy space. Attackers may try to circumvent DDAS either by brute-force guessing pointer values or derandomizing the memory layout. These techniques have been used to bypass randomization-based defenses like ASLR [23, 24, 25, 26, 89]. In this section, we analyze the effectiveness of these methods to circumvent DDAS.

4.7.3.1 Brute-Force Attacks

Given a leaked pointer and knowledge of uninflated relative distance information from the program binary, an attacker can attempt to brute-force guess a DDAS pointer for their desired target. We use a hypergeometric distribution to model how long it takes to accurately guess the target pointer when the system is re-randomized every 50 ms (*i.e.*, the attacker has a set number of guesses until the system re-randomizes and all prior attempts are invalidated). To be conservative in our analysis, we assume an attacker's guess results in a delay only when it accesses an undefined memory location, triggering an immediate security exception and stalling execution for 68 μs for exception handling [165]. We assume these security exceptions are taken and *ignored*, allowing the attack to continue uninterrupted.

For basic DDAS-R, 61 hours of continuous probing is required to correctly locate a code gadget that is 4 kB away in the original binary. For table-based DDAS-R, 86 and 87 hours is required for a 2k and 32k-entry configuration, respectively. To realistically capture relative distances, we profiled indirect jumps in the first 3 billion instructions of the SPEC benchmarks [151, 152]. More than 68 days of probing is required to guess a jump that is the average length in the *first quartile* (75.7 kB) for both table-based configurations. Probing time increases alongside target relative distance increases, with jumps in the fourth quartile taking thousands of years for all configurations.

4.7.3.2 Derandomization

Timing side channels [25, 26] and memory disclosures [23, 90, 91] have been leveraged to derandomize address obfuscation protections. DDAS is resilient to timing side-channel attacks because it does not affect cache mappings, nor are DDAS addresses used to index into microarchitectural structures (*i.e.*, caches, TLB, BTB), which are the traditional venues for side-channel exploits. Furthermore, the latency of the DDAS functional unit is not key- or

data-dependent and is implemented in constant time. Thus, timing an address translation similarly does not reveal information about the existing DDAS configuration.

As side channels are not viable, the only way to derandomize DDAS is to tactfully leak pointer values. For basic DDAS, an attacker can derive the dilation amount, i, if they obtain two DDAS/VAS pointer pairs in consecutive segment, allowing them to guess pointers with an increased likelihood of success. Although finding pointers in neighboring segments is difficult, an attacker can leverage the fact that the DDAS segment size is a power of two to narrow their search space. Table-based DDAS has more spatial diversity and therefore is more resilient to memory disclosures. With n ranges per DDAS segment, this previous technique exposes only $1/n^{th}$ of valid pointers in the address space. Furthermore, uncovering $1/n^{th}$ of the address space does not provide much of an advantage for derandomizing the rest of memory as an attacker does not know the range size or distance to the next range. Despite this, we regard memory disclosures as dangerous and employ runtime re-randomization to defend against derandomization.

By re-randomizing the DDAS configuration every 50 ms, attackers have a tight window to leak pointer pairs, perform additional probes, and synthesize an exploit. Prior work similarly uses re-randomization to shield from memory disclosures [12, 57, 58, 59, 186]. Additionally, we configure DDAS to re-randomize *immediately* when a security violation is detected. Thus, attempts to derandomize DDAS by leaking multiple pointers will fail, as leaked pointers will be invalidated by probing attempts that trigger security violations. To further harden our defense against memory leaks, the DDAS architecture could be expanded to include pointer encryption [28, 81, 82, 85]. Morpheus [31] and Shuffler [12] leverage encryption on top of address obfuscation to make leaked pointer values completely unusable. Obfuscating the virtual address space [5] or the program binary [82] is also viable, as derandomization attempts require knowledge of VAS pointers.

4.7.4 Performance-Security Trade-offs

In this work, we constrain the parameters S_{ddas}, r, and i to be a power of two to make architecture-level optimizations in the functional unit. When removing these optimizations, the DDAS system has an increased keyspace, making it potentially more secure. Intuitively, it is harder to brute-force guess the value of keys like S_{ddas} in an unconstrained system. Derandomization becomes more challenging as attackers can no longer leverage the fact that keys are a power of two to constrain their search space. Yet, the relative entropy of the defense only offers a slight improvement, at 57.5 bits for table-based, 2k-entry DDAS compared to 55-bits previously. This unconstrained variant also exhibits a 99.5% likelihood of

dilation for all jumps. However, this system incurs intolerable slowdowns. An unconstrained version of table-based, 2k-entry DDAS would have 42 cycles of exposed latency (20 cycles per divide or modulo operation), compared to 2 cycles previously. At 50 ms re-randomization, this configuration incurs over 100% slowdowns over the baseline, which we regard as an intolerable performance degradation.

4.8 Discussion and Limitations

DDAS does not protect against attacks on the data plane, such as Data-Only Programming [64], Block Oriented Programming [195], and Control-Flow Bending [69]. Since these exploits do not require code pointers, DDAS as presented in this chapter cannot stop this class of attacks. However, obfuscating the location of data objects still serves as an effective defense. To apply DDAS to the data segment, data pointer arithmetic must be maintained for program correctness. This implies that either: *i)* Dilation had to be undone for data pointer arithmetic, or *ii)* Undefined memory locations cannot be placed within data structures (*e.g.,* arrays). If the former case was adopted, DDAS would protect against attacks that compute data pointers offline and inject them via a malicious payload. However, attacks that hijack of instructions to perform data pointer arithmetic in the victim program would not engage the defense. The latter case does not work with our current dilation scheme, as it has no knowledge of memory contents and automatically inflates at a fine (4 B) granularity. DDAS and DDAS-R can be adapted for data pointers with additional work.

The DDAS architecture can be layered with existing techniques to provide greater system security and protect against memory disclosures. Although a single leaked pointer does not undermine DDAS entirely, as it does for ASLR, multiple pointers can be used to derandomize a small portion of the address space. Re-randomization provides some defense to memory leaks, as any leaked pointers expire after a fixed amount of time. To further harden our defense against memory leaks, DDAS could be combined with a form of pointer encryption [28, 81, 82, 85] to make leaked pointer values completely unusable. Morpheus [31] and Shuffler [12] take a similar approach and leverage encryption on top of address obfuscation to become more resilient against memory leaks. While these techniques serve to make leaked DDAS pointers unusable, obfuscating the virtual address space is also useful since derandomization attempts require knowledge of a DDAS and VAS pointer pair. Thus, DDAS could also be combined with existing techniques that obfuscate the virtual address space, such as ASLR [5], or the randomize the program binary [82].

While RISC-V lends itself to ease of implementation, DDAS is also alluring for other architectures. Consider ROP attacks on x86. Since instructions are variable length and

the code is incredibly dense, code gadgets can be formed by jumping midway through an existing instruction. By changing the configuration of DDAS, we could effectively inflate at a byte-granularity, adding dilation within instructions themselves to protect against the use of partial code gadgets. Fortunately, the other predominant 64-bit architectures in use today, *i.e.*, x86-64 and ARM AArch64, only implement 48-bits of virtual address, permitting opportunities for significant address space dilation in these architectures. For 32-bit architectures, we cannot exploit unused bits of the virtual address space, so this defense is less practical. Additionally, there is limited entropy in a 32-bit space.

4.9 Related Work

The work related to address obfuscation protections is extensive. Here, we restrict our comparisons to the defenses that are most relevant, *i.e.* those that employ either runtime re-randomization or in-memory traps. DDAS leverages hardware support to guarantee the necessary efficiencies while providing 63 bits of absolute entropy and 55 bits of relative entropy. With runtime overheads of 1.07% at a 50 ms re-randomization rate, DDAS outperforms many of its counterparts. For defenses with similarly low overheads, DDAS offers increased security via finer grained, higher entropy protections.

Randomization of Absolute Addresses. Defenses have employed re-randomization to fortify Address Space Layout Randomization [5] against memory disclosures. These defenses re-randomize the displacement of the code segment in response to events that are indicative of an attack, like system calls [57] or process forks [58], or at regular intervals [31]. TASR [57] has high performance overheads of 30-40%. RuntimeASLR [58] performs better than DDAS, with 0.5% overheads, but does not obfuscate relative distances between code objects. Morpheus [31] leverages hardware to achieve fast address obfuscation, with 0.9% overheads at 50 ms randomization. However, Morpheus also does not randomize relative distances and thus can be derandomized by a single pointer disclosure.

Randomization of Relative Addresses. Permutation-based defenses are the leading approach to randomize relative distances between code objects. However, unlike DDAS, the entropy of this technique correlates with code size as only a fixed number of permutations exist, providing minimal protection for short functions. Remix [59] employs periodic re-randomization and is comparable to DDAS with a 2.8% overhead. However, Remix only randomizes relative distance within functions, not between them. DDAS and DDAS-R are able to achieve more entropy than these works and randomizes both intra-function and inter-

function relative distance. We are only aware of two defenses, Shuffler [12] and Mixr [186] obfuscate the relative distance *between* functions and re-randomize at runtime. However, these defenses suffer from higher overheads than DDAS, at 14.9% or 225% respectively, and have lower entropy (much less than 30 bits).

In-Memory Traps. In addition to code diversification, some defenses use traps to probabilistically detect attacks that probe the code segment. CodeArmor [190] is a software technique that instruments the program binary with "honey gadgets", similar to undefined memory regions in DDAS. Attackers that mistakenly forge code pointers to these gadgets trigger 3 security exceptions on average before reaching their target. Using the same metric, an attacker would trigger over 100,000 exceptions in a DDAS system experiencing average dilation before reaching a code gadget. CodeArmor achieves low overheads, at 6.9% on SPEC CPU2006, but DDAS is able to provide overheads well below 5% with hardware support.

4.10 Chapter Summary

To combat code reuse attacks on the control plane, we deter the manipulation of indirect jump targets by increasing the uncertainty of the code segment. We introduce the Displaced and Dilated Address Space, a superimposed address space where all code pointers in the program are expressed. We displace the address space by a 64-bit key to obfuscate *absolute* addresses. Additionally, we leverage the vast unused portions of the virtual address space to dilate memory by over 250,000 times, obfuscating *relative* addresses. We inject large, undefined memory regions between instructions that result in an immediate security exception when accessed. Finally, DDAS code pointers are derandomized in the pipeline before use, eliminating performance impacts on the memory system.

Compared to prior work, our defense achieves greater entropy in the control plane (63 bits for displacement, 55 bits for dilation), which attackers must overcome without touching the undefined memory locations that make up 99.996% of the address space on average. Additionally, with runtime re-randomization, this challenge is coupled with a time limit of 50 ms to synthesize an attack. By leveraging hardware support, we are able to implement this defense while keeping performance overheads well below 5%.

CHAPTER 5

Redesigning Code and Pointer Protections to Use Encryption

With Morpheus and DDAS, the absolute and relative addresses of code pointers are protected in the pipeline. While powerful, these techniques are less resilient to derandomization when compared to encryption and therefore require complex microarchitectural support for run-time re-randomization. In this chapter, we revisit the original Morpheus design to leverage encryption more comprehensively. In Morpheus II [33], we bring code and pointer encryption from memory into the pipeline. Since encryption is more resilient to attack, we discard the other layers of Morpheus, eliminating the invasive microarchitecture changes without compromising security. Morpheus II's strength at protecting information assets was tested in DARPA's recent industrial-grade bug bounty exercise where, over three months, more than five-hundred attackers were unable to break into a vulnerable application running atop the Morpheus II architecture [34]. The success of Morpheus II demonstrates the applicability and promise of encrypted architectures and motivates the consideration of this approach more broadly. In this chapter, we discuss the design and implementation of Morpheus II, including how this architecture provides durable protections with relatively low overhead.

The research presented in this chapter appears in:

[2] Lauren Biernacki, Austin Harris, Tarunesh Verma, Sharad Malik, Todd Austin. ***"Encrypted Architectures for Protecting System Information Assets: The Morpheus Case Study,"*** Communications of the ACM (CACM), 2023. *Under Submission*

[33] Austin Harris, Tarunesh Verma, Shijia Wei, Lauren Biernacki, Alex Kisil, Misiker Tadesse Aga, Valeria Bertacco, Baris Kasikci, Mohit Tiwari, Todd Austin. ***"Morpheus II: A RISC-V Security Extension for Protecting Vulnerable Software and Hardware,"*** 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2021. DOI: 10.1109/HOST49136.2021.9702275.

5.1 Introduction

Morpheus' complex ensemble of defenses requires significant architectural modifications yet provides limited protection for pointers in the pipeline. We redesign Morpheus to leverage encryption more comprehensively. In ***Morpheus II*** [33], we bring code and pointer encryption from memory into the pipeline. Since encryption is more resilient to attack, we discard the other layers of Morpheus, eliminating the invasive microarchitecture changes without compromising security.

Morpheus II thwarts remote code execution (RCE) attacks by protecting code and code pointer information assets with strong encryption. Instead of configurable run-time churn, Morpheus II only changes encryption keys when a program starts or an exception is encountered, greatly simplifying the implementation of churn. We implemented Morpheus II on a modified RISC-V Rocket [196] core prototyped on Xilinx FPGAs. Our redesign of Morpheus achieves less than 2% area overheads and negligible runtime slowdowns.

Morpheus II was deployed on Amazon Web Services (AWS) as part of the DARPA FETT bug bounty program [197]. In this industrial-grade security analysis, a vulnerable COVID-19 medical database with mock data was running on a FreeRTOS web server The Morpheus II deployment engaged over 500 researchers for three months, all of whom received knowledge of the system implementation and potential vulnerabilities. None were successful at infiltrating Morpheus II [34]. The success of Morpheus II demonstrates the applicability and promise of encrypted architectures and motivates the consideration of this approach more broadly. In this chapter, we discuss the design and implementation of Morpheus II, including how this architecture provides durable protections with relatively low overhead.

Morpheus II is guided by our observation that the vast space of security attacks is based on a much smaller set of vulnerabilities, which, in turn, are exploited by attackers through understanding an even smaller set of system information assets. By protecting the information assets required by control-plane attacks, Morpheus II can provide broad coverage across existing exploits and maintain durability even in the face of future attacks based on these assets. Further, by leveraging hardware, we can protect information assets efficiently, creating a performant defenses that protects assets systematically, regardless of program-level semantics. Morpheus II's performance in the FETT challenge demonstrates the value of this defense approach.

5.1.1 Chapter Contributions

This chapter makes the following contributions:

- We present the Morpheus II Secure Architecture—a RISC-V-based processor that is designed to stop control-flow attacks by encrypting and re-randomizing code and code pointers at runtime.
- We discuss the implementation of the Morpheus II encrypted code and code pointer defenses atop a RISC-V Rocket core [196].
- We discuss Morpheus' performance as part of the DARPA FETT bug bounty program [197], where it was attacked by over 500 researchers for a period of three months. None of the attackers were successful in infiltrating the vulnerable database running on our protected Morpheus II system.
- We evaluate our design and find that the Morpheus II implementation achieves negligible performance overheads and less than 2% area overheads by leveraging simple hardware modifications.

The remainder of this chapter is organized as follows. Section 5.2 discusses Morpheus' general approach to protecting code and code pointers. In Section 5.3, we detail the implementation of Morpheus II on a RISC-V processor before discussing the FETT implementation in Section 5.4. Sections 5.5 and 5.6 discuss the performance and security of our architecture, including details of the bug bounty challenge. Finally, Sections 5.7 through 5.9 conclude this chapter.

5.2 Redesigning Information Asset Protections in Morpheus II

Guided by our insights about protecting information assets, we designed the ***Morpheus II Secure Architecture*** [33].[1] Morpheus II combines encryption, isolation, and randomization mechanisms to protect two information assets common across remote code execution (RCE) attacks—***code pointers*** and ***code***. Morpheus II encrypts code pointers and instructions, changes them rapidly, and protects related cryptographic keys. Cryptography protects code pointers and instructions from direct leakage (*e.g.*, reverse engineering), whereas frequent re-keying helps insulate these assets from side channels. Interestingly, code encryption implicitly hides the location of code objects and organization of data variables, as these assets are typically discovered through reverse engineering the program binary. Together,

[1] In this chapter, we detail the second generation design of Morpheus [33], which builds on the initial design proposed in [31].

these mechanisms protect against RCE attacks that manipulate control data to hijack victim control flow.

Morpheus II withstood three months of penetration testing in DARPA's Finding Exploits Through Tampering (FETT) bug-bounty program [34]. This success motivates the broader consideration of defenses that explicitly target information assets. Below, we discuss Morpheus II's conceptual approach to protecting information assets. In Section 5.3, we detail the implementation of Morpheus II on a RISC-V processor before discussing the FETT program in Section 5.4.

5.2.1 Security Goals

Our goal is to obstruct the acquisition of code and pointer implementation information to prevent the synthesis of control-flow attacks. Specifically, we aim to prevent the attacker from learning enough information about the victim's code and pointers to create a specific sequence of software inputs that, when provided to the victim application, without any assistance from hardware modifications (*e.g.*, Trojans), triggers a vulnerability to redirect program control flow. Specifically, we protect the following domains of implementation information:

- ***Code:*** An instruction, including the opcode (*e.g.*, `JALR`) and operands (*e.g.*, `a0`).
- ***Code Pointers:*** The stored address of a code object, including function pointers, return addresses, and any addresses in memory (*e.g.*, jump tables)

To achieve this goal, we encrypt the representation of code (`I`-8) and code pointers (`I`-6). We do not aim to prevent the creation of attack payloads that do not require knowledge of code or pointer information.

5.2.2 Threat Model

Morpheus protects against remote code execution (RCE) attacks, a subset of control-flow attacks that abuse memory errors to manipulate code pointer assets. Specifically, Morpheus II seeks to thwart attacks where a trusted but vulnerable victim program processes untrusted inputs that exploit memory errors to hijack the control-flow of the victim. This class of attacks includes return-to-libc, return-/jump-oriented programming, counterfeit object-oriented programming, and other attacks that use corrupt code pointers or inject code and hijack control flow. Yet, Morpheus's focus on protecting information assets reaches broader than this primary goal. Morpheus's always-on encryption prevents the disclosure of code

and pointers through cold-boot attacks and other information leakage mechanisms and has the possibility to impede microarchitectural side-channel attacks. The Morpheus II design places trust in the Morpheus II processor core, compiler, and a small amount of loader code.

Non-Goals. Morpheus II does not protect systems from denial-of-service attacks, fault injection, modifying the boot sequence, or tampering with the random number generator. This instantiation of Morpheus also does not prevent non-control data attacks like data-oriented programming that corrupt data pointers (as opposed to [31]).

5.2.3 Encrypting Information Assets

Morpheus II encrypts two classes of information assets—code pointers and code—to defend against remote code execution (RCE) attacks. RCE attacks manipulate program execution by overwriting code pointers (*e.g.*, return addresses) to redirect the victim program to execute the malicious code. These attacks may redirect execution to existing code (*e.g.*, return-to-libc [37] or return-oriented programming [198]) or new code maliciously injected into the victim's memory (*e.g.*, buffer overflow on the stack). The ability to identify code sequences and overwrite code pointers is the key to the successful execution of RCE attacks. Morpheus II stops these exploits by encrypting the code and code pointers needed by attackers.

Encrypted Code Pointers. In an RCE exploit, an attacker overwrites, forges, or manipulates a code pointer to redirect program execution. With Morpheus, all code pointers are encrypted all the time, including in the registers, caches, and DRAM. Morpheus II considers *any* stored address of a code object to be a "code pointer," including function pointers, return addresses, and any addresses stored in memory (*e.g.*, jump tables). Encrypted code pointers prevent the attacker from directly forging code targets, as generating a valid control target requires knowledge of the encryption key. Guessing control-flow targets is possible but will likely result in a program crash due to the garbling property of encryption. Further, strategically manipulating existing pointers requires knowledge of the code layout, which is hard to obtain when both pointers and code is encrypted. Pointer replay attacks are possible, as discussed in Section 5.7, but difficult due to Morpheus II churn (§5.2.5).

Encrypted Code. RCE exploits either directly inject instructions into victim memory or search through victim memory to locate instructions of interest (*i.e.*, code gadgets). By always encrypting code, it becomes very challenging for attackers to identify code gadgets or synthesize new code for injection, since both would require knowledge of the encryption key.

5.2.4 Isolating Encryption Keys

The cryptographic keys used for encrypting code pointers and code warrant the utmost protection, as they pose a centralized point of weakness that could compromise the entire system. To protect encryption keys, Morpheus II employs isolation, stowing all keys in protected storage (similar to Microsoft's Pluton processor [199]). This key storage is only accessible by hardware or privileged software, as necessitated to perform key exchange, thereby keeping encryption keys away from the attackers' reach.

5.2.5 Preventing Leakage with Randomization

Code and code pointers encryption strips attackers of the information assets needed to synthesize RCE attacks. However, even with strong cryptography, attackers can still infer some information assets using information leakage mechanisms like side channels. In Morpheus II, code and code pointers are decrypted to maintain program functionality, leaking related plaintext information through observable program traces (*e.g.*, memory access patterns and timing). Every time a code pointer is used to redirect program execution, information about the true memory address contained in that pointer is leaked via observable program traces. Likewise, every time an instruction is decrypted and executed, information about the instruction's functionality is leaked via timing and contention in the architecture. Clever attackers could use side-channel analysis or other mechanisms to learn the values of these information assets.

Morpheus II Churn. To combat information leakage, we introduce the periodic re-keying of the cryptographic defenses, termed ***churn***, to nullify information an attacker may have gained via side-channel leakage. After a churn, all instructions and code pointers in the system take on a new form, ensuring that any valuable information gathered by attackers since the last churn cycle will be lost. This approach creates high-entropy situations where attackers are confronted with a system they have never seen before and will never see again. Churn can be configured to occur periodically or in response to a particular event, such as a security violation.

5.3 The Morpheus II Secure Architecture

Morpheus II implements its protections detailed in Section 5.2 in hardware to create a relatively simple and inexpensive defense that can provide durable coverage against exploits. In this section, we detail the implementation of the Morpheus II defenses on a RISC-V

Rocket core in-order scalar processor [196] and highlight how our implementation decisions minimize performance overheads. We prototyped our design on a Xilinx FPGA to determine its hardware and runtime overheads (§5.4). This open-source processor implementation was critical for design validation by equipping us to fully specify a system for security researchers to study. Specifically, this FPGA implementation enabled us to participate in a bug bounty program, as detailed in Section 5.6.

Instruction Class	Example	Function
Arithmetic	`enc_addi rd, rs1, IMM`	`rd = enc(dec(rs1) + IMM)`
Relational	`enc_sleq rd, rs1, rs2`	`rd = dec(rs1)` $\leq$ `dec(rs2)`
PC-Relative Jump	`enc_jal rd, IMM`	`rd = enc(PC+4), PC = PC+IMM`
Reg-Relative Jump	`enc_jalr rd, rs1`	`rd = enc(PC+4), PC = dec(rs1)`

Table 5.1: **Morpheus II RISC-V ISA Extension.** Morpheus II adds three classes of instructions to support pointers: *i)* Encrypted pointer ALU operations, *ii)* Decrypted pointer relational tests, and *iii)* Decrypted unconditional jumps.

5.3.1 Hardware Architecture

Below, we discuss the implementation details of the Morpheus II encryption (Section 5.2.3), isolation (Section 5.2.4), and randomization (Section 5.2.5) protections, in that order.

5.3.1.1 Encrypted Code Pointers.

Morpheus II protects code pointers by encrypting them in memory and in the register file using a ***twelve-round Simon cipher*** [200, 201]. This version of Simon was chosen because it could be implemented in a *single cycle* on our 50 MHz processor, minimizing the performance impact while providing strong encryption. Hardware extensions to support the encryption of code pointers are shown in Figure 5.1.

Encrypting code pointers is challenging because the program counter (PC) must be preserved to maintain program functionality and correctness. We achieve this by using a RISC-V ISA extension that implicitly decrypts code pointer arguments prior to forwarding them to the PC. This ISA extension, shown in Table 5.1, enumerates all possible instructions that consume or produce code pointers.

The ISA extension preserves functionality for jump instructions that redirect control flow. To support jump and link register (`enc_jalr`) instructions, which redirect control to an encrypted code pointer, Morpheus II decrypts the code pointer in the execute stage of the pipeline ① then increments it by the instruction's immediate. The resultant target address

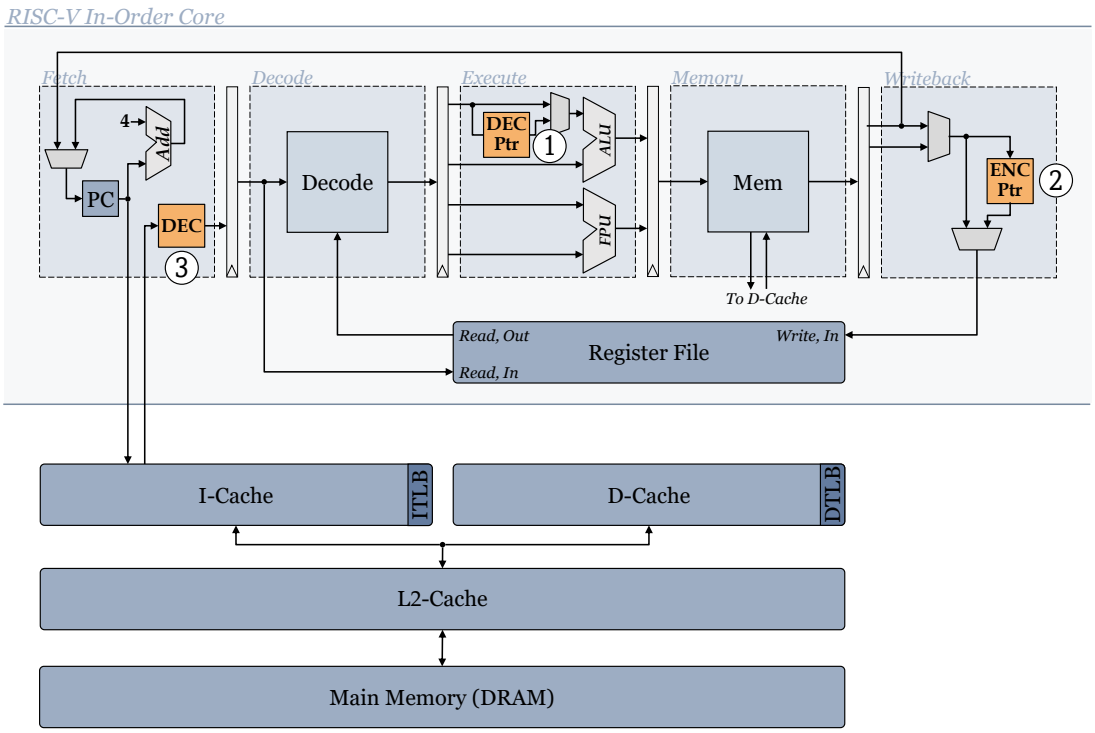


Figure 5.1: **Morpheus II Architecture.** The Morpheus II secure architecture extensions serve to maintain the correctness of program execution when code pointers and code are encrypted. The Morpheus II extensions decrypt the target address for register-relative jumps ① and encrypt the return address for all control-flow redirections ②. Instructions are decrypted prior to the decode stage ③. These ciphers latencies fit within the current pipeline stages, minimizing runtime overheads. Compared with the previous Morpheus design (Figure 3.4), Morpheus II greatly reduces pipeline modifications and memory touchpoints.

is then forwarded to the PC. Jump and link (`enc_jal`) instructions, which do not use code pointers, are not modified in Morpheus II as immediates and the PC are not encrypted. In addition to computing the target address, `enc_jal` and `enc_jalr` instructions store the return address of the control transfer (PC+4) into the destination register. Morpheus II encrypts this return address prior to writing it to the destination register ②, maintaining always-encrypted code pointers.

Finally, encrypted code pointers can be used for arithmetic and relational instructions to compute new pointer values or make comparisons. Our ISA extension accommodates code pointer arguments for these instruction classes by automatically decrypting inputs for pointer arithmetic and relational operations ①, and re-encrypting results for always-encrypted pointer ALU operations ②.

5.3.1.2 Encrypted Code

Morpheus II also protects code information assets, as discussed in Section 5.2.3, by encrypting all instructions using a twelve-round Simon cipher [200, 201]. The Morpheus II compilation process is succeeded by a small software routine that encrypts the ELF binary in preparation for the Morpheus II architecture. Since we encrypt the program binary after compilation, this would not be suitable for just-in-time (JIT) compilation. Extensions to support the encryption of instructions are shown in Figure 5.1.

Encrypting code is challenging because the decode stage of the pipeline needs access to unencrypted instructions. Thus, Morpheus II decrypts instructions after they are read from the I-cache, prior to being placed in the fetch queue ③. This design allows the instructions remain encrypted in memory while preserving the decode stage functionality. Furthermore, the decryption of instructions is off the critical path, minimizing performance overhead.

5.3.1.3 Key Management.

Key management, in terms of both ensuring the security of the encryption key and the efficiency of key exchange, is critical. The Morpheus II ***runtime encryption key*** is stored within a secure, isolated hardware register and is only read during the fixed-latency encryption and decryption operations within the pipeline (Figure 5.1). These registers are not accessible to software-based addressing and are only read/written implicitly through the Morpheus II interface (*e.g.*, `enc_jalr`). To simplify the implementation of our FPGA prototype, Morpheus II encrypts the target binary using a fixed ***boot-time key*** stored within the Morpheus II architecture and known by program developers.[2] At load time, the target binary is re-encrypted under a newly generated runtime encryption key, reconstituting all instructions in the code segment and all code pointers in the data segment. The logic used to re-encrypt code at load time is also used to support churn, as described below. Both the boot-time and runtime key are stored within secure registers and are only utilized during the fixed-latency cryptographic operations in the pipeline.

5.3.1.4 Preventing Leakage with Randomization (Churn).

In Section 5.2.5, we introduced the concept of churn, or the re-keying of code pointer and code encryption, to thwart information leakage. Implementing churn is challenging because locating and re-encrypting all code pointers and instructions in memory requires a significant

[2]Alternatively, this key could be communicated to the architecture via a hardware-backed public-key exchange protocol.

amount of computation.[3] To overcome this challenge, Morpheus II takes advantage of the existing boot process which reconstitutes code pointers and code from the program binary and encrypts them under a new runtime encryption key. When Morpheus II detects an exception (*e.g.*, segmentation fault or misaligned instruction fetch), the system is quickly warm-booted. A new runtime encryption key is generated, and all code and code pointers are reconstituted under this new key. All code and code pointer information assets now take a different form than their last instance.

This instantiation of churn cannot re-randomize encrypted assets under running programs, instead stalling the program during churn updates. However, it avoids the invasive microarchitectural changes of adding a physical churn unit, as done in the original Morpheus design [31] (Chapter 3).

5.3.2 Compiling Protected Applications

Program binaries must engage Morpheus's protections by calling the encrypted code pointer instructions contained within the Morpheus II ISA extension (Table 5.1). This process is facilitated by the Morpheus II LLVM compiler. The Morpheus II compiler detects code pointer provenance and instructions that consume code pointers (*e.g.*, `addi`), signaling a backend LLVM pass to emit instructions from the Morpheus II ISA extension accordingly (*e.g.*, replace `addi` with `enc_addi`). Code pointer provenance can occur in two ways: *i)* The code address is built during runtime via a sequence of instructions (`lui`/`addi`), or *ii)* The code address is statically initialized at compile time and stored as a constant in the text segment. The Morpheus II compiler supports code pointer generation via (i). Supporting code pointers generated via (ii) is possible but was not necessary because our baseline RISC-V LLVM compiler does not statically initialize code pointers.[4] The Morpheus II compiler appends these code address generation sequences with an instruction that encrypts the computed pointer before storing it to memory. This compilation process is succeeded by a small software routine that uses the Simon block cipher [201] to encrypt instructions in the code segment under the boot-time key. Morpheus's protections can apply to both user and kernel code, given that the binaries are produced by our software toolchain. In our implementation, we include a protected RISC-V port of the FreeRTOS operating system kernel [202] and the Newlib C library.

[3]The original Morpheus II design proposed in [31] used tagged memory to locate code and code pointers during churn. Tagged memory is complex to implement and incurs significant hardware area overheads.

[4]The Morpheus II compiler was built atop a defeatured version of RISC-V LLVM that only supports code pointer generation via (i). At the time, RISC-V LLVM had limited capabilities, including no support for jump table lowering, motivating this choice.

Software Accommodations for Encrypted Code and Code Pointers. Expert system designers may intentionally manipulate code or code pointers for their benefit. Examples include self-modifying code and pointer alignment calculations. Traditionally, such manipulation is possible because the abstraction layers are imperfect, allowing compilation- or ISA-level semantics to be accessible from high-level programming languages. However, running these programs on the Morpheus II architecture will lead to unintended behaviors or program crashes, as code and code pointers are now encrypted. Despite these concerns, we have found cases of programs manipulating code or pointers to be infrequent and trivial to resolve. During our implementation, only three lines of code in the FreeRTOS assembly files needed to be changed to accommodate the Morpheus II protections.

5.3.3 Differences From the Original Morpheus Design

Our FPGA-based implementation differs from the original Morpheus design (Chapter 3) in three notable ways. First, there is no encryption of data pointers. While the Morpheus II implementation contains the necessary hardware to support data pointer encryption, this defense requires additional compiler modifications to identify data pointers. We suspect that data pointer encryption would introduce greater runtime slowdowns due to latencies involved with data pointer arithmetic. Second, Morpheus II forgoes microarchitectural tagging, instead using an explicit ISA extension to communicate types (*e.g.*, code pointers) to the architecture. Tagged memory is expensive and requires significant changes across the microarchitecture. Other hardware security solutions are also moving away from the use of tagged memory [27]. Finally, the FPGA implements churn only when a security violation or exception is detected, rather than at a fixed time interval. When coupled with accurate security detection mechanisms, this implementation of churn is more efficient, lessening the burden on program runtimes.

5.4 Experimental Methodology

Morpheus II was implemented on an Xilinx UltraScale Plus VCU118 FPGA as an extension to the RISC-V Rocket core [196], built with the design parameters detailed in Table 5.2. Our VCU118 FPGA implementation allowed us to analyze the power and area overheads of the Morpheus II design in more detail. While an FPGA-based implementation is not as fast or power efficient as a full-custom ASIC implementation, it does offer a hardware realization of the design under a reasonably short timeline and cost. We measure performance, area and power costs relative to a baseline design on the same FPGA target. The trends captured in

these measurements are reflective of an ASIC-based deployment, to a first order, as FPGA-based overheads are larger than would be seen in an ASIC design. In addition, our detailed FPGA-based deployment enabled a full-fledged industrial-grade security analysis as part of DARPA FETT bug bounty program [34], as discussed in Section 5.6. For additional evaluations of the Morpheus II architecture, please refer to [33].

Architecture	RV32IMAC, 50 MHz
Core	In-order, 5-stage pipeline
Caches & Memory	4kB I-cache, 4kB D-cache, 2GB DDR4
Devices	Ethernet, UART
Morpheus II Crypto	12-Round Simon cipher, 1-cycle

Table 5.2: **Morpheus II FPGA Build Specification.**

Compiler Support and Benchmarks. We evaluated Morpheus II on a suite of benchmarks including Coremark [203], MiBench [150], and our SQLite Mock Medical Database application used for the DARPA FETT program. All benchmarks were compiled using the Morpheus II LLVM compiler and encrypted accordingly. Benchmarks exhibited mostly PC-relative jumps (`enc_jal`) for function calls and register-relative jumps (`enc_jalr`) for returns. Our workloads only rarely exhibited arithmetic or relational operations on code pointers, as such logic is not typical.

5.5 Experimental Results and Evaluation

To measure the performance impacts of Morpheus II, we ran experiments against the MiBench [150] and Coremark [203] workloads. We also measured the power and area overheads of the Morpheus II FPGA extensions using Xilinx Vivado. These experiments are discussed below.

5.5.1 Runtime Performance Overheads

In our evaluated workloads, the observed runtime performance overheads due to the Morpheus II defenses were negligible ($\ll$1%). Slowdowns due to the Morpheus II protections occur when: *i)* A register is decrypted for use in a indirect (register-relative) jump, and *ii)* An instruction that is not in the instruction queue is decrypted on a branch mispredict. In both cases, execution is delayed for one clock cycle due to the latency of the Simon cipher. Morpheus II churn constitutes additional overheads, but only under a security attack.

Benchmark	Morpheus II Slowdown
Coremark (2000 Iterations)	0.045%
adpcm decode	-0.0228%
adpcm encode	-0.0239%
aes	-0.0018%
basicmath	-0.0009%
blowfish	-0.0003%
crc	-0.0089%
fft	-0.0098%
limits	0.0587%
qsort	-0.0046%
randmath	0.0016%
rc4	-0.0042%
MiBench Avg	-0.0022%

Table 5.3: **Performance Slowdown.** This table lists the recorded slowdown benchmarks running atop Morpheus II compared to an unprotected baseline. Differences between the two architectures are negligible, with a majority of applications actually experiencing a slight speedup (<0.01%) on Morpheus. This speedup can be attributed to system noise or the fact that Morpheus' re-randomization prefetches code and pointers into the cache.

Table 5.4 compares the performance of Morpheus II with the baseline system on SQL queries run on M3DB, a synthetic database containing information on COVID-19 patient outcomes. The platform generates the database with random data entries at boot time and receives and responds to SQL queries on this database via a REST interface. The table shows minuscule differences between the average execution times for a few sample queries on the baseline system and Morpheus II. The differences are within the bounds of run-to-run variations, showing that Morpheus II does not add any performance penalties for the database application.

5.5.2 Power and Area Overheads

The power and area overheads of the Morpheus II architectural modifications were measured using Xilinx Vivado over the Rocket core baseline [196]. These measurements are representative of the amount of overhead this design is experiencing due to addition of Morpheus II defenses. The power increase due to the extra logic and ciphers was only 0.21% for the entire Rocket Core design, including the DRAM controller and the XDMA PCIE bus controller. Area overheads were uniformly low, at 1.29% for LUT (logic) resources and 0.06% for registers.

Database Query	Average Execution Time (seconds)			95% Confidence Intervals	
	Baseline	Morpheus II	Δ	Baseline	Morpheus II
`HELP Page`	0.0244	0.0254	0.0010	0.1047	0.161
`avg(Recovered)`	1.5293	1.5648	0.0355	0.0248	0.0728
`avg(TestPositive)`	1.5310	1.5504	0.0194	0.0551	0.0371
`avg(RecentTravel)`	1.5382	1.5281	-0.0101	0.0265	0.0286
`avg(RecentTravel) where (gender==M")`	1.4663	1.5158	0.0495	0.0680	0.0449
`avg(RecentTravel) where (zipcode==48105 && gender==\"M\" && reqvent==1)`	1.0349	1.0662	0.0313	0.0042	0.0305

Table 5.4: **Runtime of Evaluated Database Queries.** We evaluated the performance overheads of various queries on our SQL database application deployed in the DARPA FETT challenge. The differences in execution time are within the bounds of run-to-run variations, demonstrating that Morpheus II has negligable performance impacts on the system.

5.6 Security Analysis

In designing Morpheus, our thesis was that protecting information assets provided a more durable approach to stopping known and unknown attacks. We had the opportunity to test this hypothesis through participation in DARPA's Finding Exploits to Thwart Tampering (FETT) Bug Bounty challenge [34]. In this section, we detail the FETT challenge and additional security analysis of Morpheus.

5.6.1 Qualitative Discussion of Security Against Common Attacks

Next, we describe additional modern attacks and how Morpheus II would prevent them.

ROP. To dispatch ROP gadgets, a memory corruption vulnerability is exploited to overwrite a return address with a code pointer. In Morpheus II, this pointer is encrypted, barring the synthesis of an attack. Variants of ROP that exploit other control-flow statements, like jump-oriented programming [67], also fall under this category.

JIT-ROP. JIT ROP attacks [23, 204] utilize gadgets generated by the just-in-time compiler based on attacker controlled inputs. If Morpheus II was extended to generate encrypted JIT code, an attacker would not be able to disclose the plaintext code needed to mount the attack in [23]. While [204] does not rely on disclosing plaintext code, the encrypted code pointers in Morpheus II would prevent a successful attack.

COOP. Counterfeit Object-Oriented Programming (COOP) [205] is an advanced code-reuse attack that relies on chaining gadgets from the C++ virtual function table. Morpheus II stops COOP by preventing injection of counterfeit objects (code encryption) and requiring

the attacker to forge encrypted pointers for the addresses of gadgets in existing objects (code pointer encryption).

5.6.2 Bug Bounty Deployment

In the summer of 2020, the Morpheus II secure architecture was entered into the DARPA FETT challenge [34], run by Synack, along with six additional secure CPU designs from the DARPA SSITH program [206]. FireSim [207] was adapted to provide a test harness for the challenge, enabling crowd-sourced security researchers to spin up an Amazon AWS F1 FPGA cloud instance containing Morpheus II or another DARPA SSITH secure core to attack.

All teams in the DARPA FETT program were required to provide a ***challenge application*** with known software vulnerabilities. The Morpheus II challenge application was a SQLite Mock Medical Database (M3DB) containing faux COVID-19 patient data running on a FreeRTOS web server. The real-time operating system, full network stack, and sensitive medical database were all fully protected by the Morpheus II architecture. The M3DB application accepted queries for patient data via a network-facing REST interface. As the strongest test to any defense comes under the situation where the attacker has full knowledge of the system implementation, security researchers participating in the FETT challenge were given access to Morpheus II implementation details and the software running on it, with the exception of the boot-time and runtime encryption keys. For an attack to be considered successful, researchers had to penetrate the M3DB database to modify or exfiltrate patient data. While Morpheus II was designed to defend against RCE attacks (Section 5.2.2), *any software-driven exploit that achieved this goal was allowed.*

During the FETT Challenge, the Morpheus II target engaged 535 security researchers, enduring three months of penetration attempts. Despite being the second most attacked target in the FETT program, ***the Morpheus II target was never penetrated and no exploitable vulnerabilities were discovered.*** DARPA disclosed that researchers had discovered 10 vulnerabilities across the six other targets [34]. Morpheus' exceptional performance in the FETT challenge received widespread attention in the press (*e.g.*, [208]).

5.6.3 CWE Evaluation

Morpheus II provides broad coverage of exploits and vulnerabilities by protecting two critical information assets—code pointers and code. In this section, we return to this claim and analyze the performance of Morpheus II against different CWE classes.

We derived a set of bare-metal tests that implement attacks from several CWE classes, shown in Table 5.5. Morpheus's code and code pointer encryption mechanisms were able to

CWE Class	Assets	Result
CWE-121: Stack-based Buffer Overflow	CP	**Stopped**
CWE-416: Use After Free	CP	**Stopped**
CWE-94: Improper Control of Generation of Code	C, CP	**Stopped**
CWE-190: Integer Overflow or Wraparound	C, CP	**Stopped**
CWE-257: Storing Passwords in a Recoverable Format	D	**Successful**
CWE-209: Information Exposure through an Error Message	DP	**Successful**

Table 5.5: **CWEs Evaluated on Morpheus.** This table lists CWE classes and information assets they exploit, denoted as representation of code (C), code pointers (CP), data pointers (DP), or data (D). Morpheus II did not stop CWEs that used data or data pointers, as it does not protect these assets.

successfully thwart all weakness that required access to these assets. Morpheus II failed to defend against attacks that exploited only data or data pointers, since our design does not protect these assets (*i.e.*, CWE-257 and CWE-209). However, there is nothing fundamentally difficult in protecting these additional information assets. The Morpheus II approach can be applied broadly. Hardware extensions that encrypt and obfuscate data and data pointers can prevent additional exploits, and our upcoming work is focusing on this implementation.

5.6.4 Attacking Morpheus II

5.6.4.1 Derandomization through Side Channels

Since brute-force attacks on Morpheus II cryptography are infeasible, attackers must resort to side channels to leak information. Encryption keys are stored in the hardware, and only utilized during the fixed-latency encryption and decryption within the pipeline. In addition, micro-architectural structures like caches never hold plaintext pointer values as they never leave the pipeline unencrypted. Similarly, cold-boot attacks on DRAM or disk are unsuccessful as pointer values are fully encrypted throughout the system.

5.6.4.2 Derandomization through Blind-ROP

For further security analysis, we performed our own penetration testing by exploring the utility of Blind Return Oriented Programming (BROP) [24] attacks on the M3DB application. The BROP approach is suitable for attacking closed-source proprietary services with non-public binaries. BROP requires a stack overflow vulnerability and a service which restarts after a crash (for example, a web server or a database application), both of which are represented in the Morpheus II-protected M3DB target application. The attack works by first breaking Address Space Layout Randomization (ASLR) and then finding remote

gadgets to call a `write()` function, which leaks the binary from the memory to the attacker over the network. The attacker then performs a conventional ROP attack using knowledge of the leaked binary. [24] provides further details about the various gadgets required for a successful BROP attack.

For the baseline system without Morpheus II protections, scanning for a particular ROP gadget from known start of code range until known start of stack takes about 57,500 tries. The attacker runs `curl` to send address that can overflow on stack. Given how the application is designed, it takes the attacker 1.29 seconds to determine that the attempt was unsuccessful. Moreover, each unsuccessful attempt causes a soft reboot which has a latency of 13.6s for the 50 MHz frequency of the platform. It would thus take almost 10 days to probe just for a single ROP gadget. Note that the M3DB web server supports 20 requests in parallel, but unsuccessful requests must be tried sequentially while the application performs a soft reboot. After guessing the correct address, the attacker successfully executes the gadget.

Once Morpheus II protections are turned on, the attacker must guess the encrypted address of any and all gadgets required to break ASLR, execute `write`, and then execute ROP from the full 32-bit address range. Since the encryption key changes at warm boots (re-randomizing all key assets which BROP expects to be static across reboots [24]), each attempt at an BROP will be a pure random guess into the 32-bit address space. Given that the expected number of trials (of a Bernoulli process) for a domain of size N is N trials, a successful BROP attack will require approximately 4 billion attempts *for each gadget*. With each attempt taking 14.89 seconds, one would expect an expected latency for this attack of nearly 2000 years! In addition, any attempts which crash the system would need to restart the entire probe process, and wait for the M3DB application to restart, further delaying the attempted hack.

5.7 Discussion and Limitations

The Morpheus II toolchain does not statically initialize code pointers or support just-in-time (JIT) compilation. Support for these features is possible but is not included in our current software toolchain.

Morpheus II addresses a small subset of the information asset landscape, only guarding against attacks that directly manipulate code or code pointers. Morpheus II does not defend against attacks that exploit other assets, as stated in Section 5.2.2. If an attacker has a known encrypted code pointer, Morpheus II does not obstruct a replay of that code

pointer within the same churn period (termed a pointer replay attack).[5] Without a pointer disclosure, committing a pointer replay attack is non-trivial. Knowledge of the code layout from the program binary is not enough to synthesize a replay attack since code pointers must be encrypted under the current key. If an attacker attempted to guess the key or encrypted pointer, they would likely trigger a program fault, initiating a churn cycle and re-randomizing the system. We view Morpheus II's performance in the FETT bug bounty program as significant evidence that replay attacks are challenging to commit on the Morpheus II architecture. To better guard against replay attacks or prevent long-lived keys, a periodic churn cycle could be used, as done in [31]. Code layout randomization could also be layered with Morpheus II to complicate efforts to learn code addresses from the program binary.

5.8 Related Work

Morpheus is one of many proposed defenses that use encryption or other techniques to protect information assets. Below, we detail additional related work that achieves security by protecting code and pointer information assets.

Protecting Code Contents. Defenses protect code and its contents by either encrypting the program binary or obfuscating its contents. Instruction Set Randomization (ISR) [72, 73] and related techniques [81, 82] randomize the representation of code at either compile-time or load-time to thwart attacks that leverage code injection or gadget discovery. Morpheus similarly encrypts instructions in the victim binary [31, 33], but does so with a strong cipher that is re-keyed at runtime.

Rather than obfuscating code representation, other techniques prevent code access or obfuscate code contents. Execute-no-Read (XnR) [92] restricts read access for code altogether, enforcing permissions for code pointers and data pointers to prevent software from reading code. Subsequent work [93] integrated this technique with existing mitigations that randomize jump targets (*i.e.*, [90, 91]) to efficiently mitigate both direct and indirect forms of code gadget discovery.[6] Polyverse [63] randomizes program behavior, including register usage,

[5] The original Morpheus design (Chapter 3) does prevent replay attacks since code pointers are encrypted using their effective address as a tweak.

[6] Many defenses protect both code contents and the addresses of code objects or representation of pointers, as these information assets are closely intertwined. Namely, obfuscating code contents indirectly helps protect code addresses, as addresses cannot be learned from analyzing the program binary. Further, pointer defenses can be broken if code is exposed, since direct and indirect jumps in the program binary leak plaintext pointer values. Thus, these techniques are often combined.

function locations, import tables, and other targets. Each binary produced by Polyverse is unique yet preserves the semantics of the intended program, effectively obfuscating critical attack assets like code layout and behavior.

Encrypting Code Pointers. Techniques like PointGuard [28], StackGhost [209], and Cryptographic Capability Computing (C3) [27] leverage encrypted code pointers or data pointers to insulate applications from from attack.

PNS [210] implements strong-encryption of code pointers, like Morpheus II, utilizing special instructions for encryption and decryption. Morpheus II instead implements encrypted variants of existing address generation and jump instructions, removing the extra overhead of instructions specifically for encryption and decryption. However, Morpheus II requires recompilation while PNS can implement partial protections using binary instrumentation. Concurrent work ZeRØ [211] uses special instructions to provide code and data pointer integrity with negligible performance overhead. Compared to Morpheus II, ZeRØ lacks code protections and has higher area and power overheads due to cache modifications needed to track pointers.

Protecting Memory Pointers and Addressing Defenses have also randomized the location of code and data objects as a mechanism to protect pointers. There is a significant body of work focusing on randomizing memory locations to obfuscate the addresses of code and/or data objects. The seminal work on Address Space Layout Randomization (ASLR) [5] shifts the location of the stack, heap, and code in memory at load time, hardening the acquisition of related information assets. Subsequent ASLR-variants sought to re-randomize the location of object more dynamically, such as after every kernel I/O operation [57], processes fork [58], or periodically [12, 31]. Other approaches have taken a more fine-grained approach to memory randomization by reordering basic blocks [59], functions [12], and stack contents [95], or by directly inflating the address space to obfuscate relative distances [32].

Related protections that target vulnerability classes focus on guaranteeing pointer integrity or enforcing memory safety (*i.e.*, detecting when pointers are corrupted) [60, 76, 212, 213]. Under these defenses, pointer assets are still readable but cannot be overwritten, thereby preventing many existing attack avenues that use pointer manipulation. Control-Flow Integrity (CFI) [61] further restricts the usage of pointers in constructing exploits as it validates all control transfers against the program's control-flow graph. From an information asset perspective, CFI restricts the domain of valid code pointers for a particular jump instruction to that specified by the programmer. CFI's high performance overheads have motivated coarse-grained variants that proved to be susceptible to attackers [62].

5.9 Chapter Summary

Hardware defenses are uniquely positioned to protect critical information assets exploited by attackers. With Morpheus, we demonstrate the power of hardware defenses to provide broad coverage over the attack landscape through a few simple mechanisms. Morpheus II defends against remote code execution attacks by encrypting and periodically re-randomizing code pointers and code in hardware, stripping attackers of the information needed to synthesize an exploit. To test the validity of design, we built and deployed Morpheus II on an FPGA-based system that was attacked by over 500 security researchers, all of whom were unable to infiltrate our design [34]. Morpheus II accomplishes this with negligible performance impact and less than 2% area overheads, establishing it as a lightweight but durable security mechanism. Morpheus II demonstrates the ability of hardware defenses to provide coverage of the attack landscape by encrypting a few critical information assets. Further, the success of Morpheus II in the FETT challenge demonstrates the applicability and promise of encrypted architectures more broadly.

CHAPTER 6

Interlude: From Security to Privacy

In the second half of this dissertation, we pivot from protecting code and pointers to protecting sensitive data. This shift is a confluence of multiple considerations, which we summarize below. The following chapters continue to embody our focus on protecting information assets, except we now turn to defend the asset that many attacks aim to exfiltrate—*data*. Further, our research continues to embrace vulnerability-agnostic design, as vulnerabilities can be present in system software yet cannot compromise data privacy.

- ***Reflecting on Morpheus:*** Our approach to guarding against control-flow attacks evolved to encrypting code and code pointers in the pipeline. This design of Morpheus II was demonstrated to be powerful not only for stopping control-flow attacks but also other attacks that leverage code and pointers when it outlasted its peers in a three-month bug bounty program that engaged over five-hundred security researchers [34, 208, 214, 215]. This result inspired us to question what else Morpheus II could be used for. Morpheus [31] originally categorized memory values into four domains: code, code pointers, data pointers, and data. While we had studied the first three, we had not yet considered how encryption protections could be applied to data values.

- ***The Rise of Homomorphic Encryption:*** As Morpheus II is engaged in the FETT challenge, homomorphic encryption (HE) is garnering more and more attention from the security community. HE is a class of cryptographic algorithms that preserve operations performed on ciphertexts after decryption. HE imbues systems with the ability to compute *directly* on ciphertext values, thereby never exposing plaintexts or keys to untrusted software. As a result, HE is regarded as the holy grail of data privacy. However, it does not come without cost. HE has extensive performance overheads that often exceed many orders-of-magnitude. These exorbitant overheads are the focus of a large body of ongoing research. Yet, potential alternatives also warrant consideration. As we watch the Morpheus II encrypted architecture excel, we see great potential for it to be applied to privacy to build an efficient alternative to HE.

- ***Resilience from Side Channels:*** While Morpheus and DDAS provide strong security guarantees against control-flow attacks, their designs inherently leak information through side channels. These architectures maintain a native, unprotected program counter (PC). Code pointers are decrypted or translated before being loaded into the PC, which is then used to index the I-cache. Such accesses are measurable via timing side channels (*e.g.*, [101]), thereby leaking the value of the protected code pointers. Similarly, encrypted code leaks through contention for pipeline functional units and encrypted data pointers leak through D-cache lookups. Morpheus and DDAS leverage re-randomization to thwart this leakage. But, this approach is statistical and has been shown to only add one "bit of entropy" to the system [89]. Moreover, side channels that leak information via measurable state (*e.g.*, timing) are inherent to the design of modern microarchitectures. Any security defenses layered atop these vulnerable microarchitectural structures are susceptible to side-channel attacks. To ensure defenses are insulated from side-channel attacks, we must build resilience from the ground up, rather than try to remediate it later. Custom microarchitecture designs enable this approach.

- ***Considering Trust in Hardware:*** While we can mimic the goals of HE with encrypted architectures, a user must ultimately trust the hardware or at least a small portion of it. We believe that trust in hardware can readily be established. Furthermore, if our design is small enough, formal verification techniques could be used to establish this trust. Current architectural approaches are very difficult to formally reason about [216]. To prove the security guarantees of a system like Intel SGX, one must model all processor features that expose registers. Not only is this infeasible, but any modifications to the architecture would invalidate this security proof. Rather, we aim to shrink the footprint, or root-of-trust (RoT), of our architectural extensions, hoping that this may lead to formal verification opportunities that will provide confidence in our hardware-based approach.

A Leap of Faith. As we reminisced on the success and trajectory of our defenses targeting control-flow attacks, we began to consider the quote that opens the introduction of this dissertation: *"Are we crazy enough to put encryption in the pipeline?"* In response, the answer was not an astounding yes but more of *"One toe on board."* Informed by Morpheus, we began to explore how trusted hardware can provide comprehensive data privacy in a wholly untrusted or vulnerable software environment. Our simple yet powerful design, termed Sequestered Encryption (SE), encrypts data using strong ciphers like AES-128 with reasonable overheads ($<2.5\times$ after optimizations). The following chapters detail our design of this architecture that encrypts data in use and our work to remediate its overheads.

CHAPTER 7

Architectural Approaches to Encrypting Data In Use

While Morpheus and DDAS provide strong security guarantees against control-flow attacks, their designs inherently leak information through side channels, as both defenses maintain an unprotected program counter (PC). Motivated to address these limitations, I began to explore how trusted hardware can protect *data* comprehensively and without side-channel leakage. We present Sequestered Encryption (SE)—a hardware technique for data privacy that sequesters sensitive plaintext data into a small hardware root of trust and encrypts this data in all external microarchitectural structures, thereby rendering secret values inaccessible to software. SE takes a similar approach to Morpheus by encrypting its targeted information asset. However, unlike Morpheus, it never exposes decrypted values to side channels or other leakage mechanisms. Thus, it does not require runtime re-randomization to thwart attacks. This design also avoids the significant implementation overheads associated with re-randomization, as we see in DDAS and the original Morpheus publication [31]. SE achieves ∼36× performance slowdowns on average compared to native execution. We find that these overheads scale proportionally with the latency of the encryption scheme, as the Simon cipher, which takes half as many cycles, incurs overheads of 19× on average. While SE is more performant than existing approaches, we see additional opportunities to optimize our design. In Chapter 8, we detail architectural optimizations that demonstrate that SE can emerge as a dynamic, expressive, and performant data privacy solution.

The research presented in this chapter has been published in:

[35] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Fitsum Assamnew Andargie, Todd Austin. ***"Sequestered Encryption: A Hardware Technique for Comprehensive Data Privacy,"*** 2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED), 2022. DOI: 10.1109/SEED55351.2022.00014.

7.1 Introduction

The task of securing private data is one of the great challenges of modern information systems. Security vulnerabilities are expected to be present in complex systems software, yet such systems are trusted to process sensitive user data. As a result, data breaches have become pervasive. In 2020, IBM estimated that public web servers have a 1-in-4 chance of being penetrated in any given year, with each breach costing an average US$3M to recover [217]. This, coupled with recent advances in data exfiltration (*e.g.*, Spectre [16]), has brought data privacy to the forefront of system design concerns.

Insulating complex systems and cloud applications from attack is challenging due to the pervasiveness of software vulnerabilities. It is simply insurmountable to adequately eliminate all software vulnerabilities, and therefore imprudent to rely on software to protect user data. Vulnerability-tolerant secure architectures like Morpheus [31, 33] show promise at maintaining system security despite pervasive software errors. However, Morpheus' bounded threat model cannot prevent attacks that require other, unprotected information assets. Thus, these attacks could still be used to exfiltrate sensitive user data.

An alternative solution to relying on security protections is to ***uphold data confidentiality, even in the presence of vulnerable or compromised software***. Effectively, this would mean repurposing secure architectures to protect data directly. Homomorphic encryption (HE) is the only present technique that achieves this goal by imbuing systems with the ability to compute directly on ciphertext values. These cryptographic schemes behave as homomorphisms between the plaintext and ciphertext spaces, preserving operations performed on ciphertext values after decryption (*e.g.*, $\mathsf{ENC}(x) + \mathsf{ENC}(y) = \mathsf{ENC}(x + y)$). An application that leverages HE can compute on data without needing access to encryption keys or sensitive plaintext values. Thus, even if the application is compromised, its data will not be exposed, as plaintext values do not exist within the system. While HE has made breakthroughs in providing comprehensive data privacy, its limited multiplicative depth has remained the main practical limitation, bounding the scheme's expressiveness, exacerbating performance overheads, and ultimately inhibiting adoption.

The escalation of data breaches and shortcomings of HE warrant alternative solutions for providing comprehensive data privacy. Trusted hardware is uniquely positioned to administer data privacy, especially in the presence of vulnerable software, as it underlies all software abstractions and can be applied systematically, regardless of program-level semantics. Furthermore, hardware techniques have seen widespread adoption for providing secure key storage, isolated execution, and code integrity (*e.g.*, Intel SGX [216, 218]), demonstrating

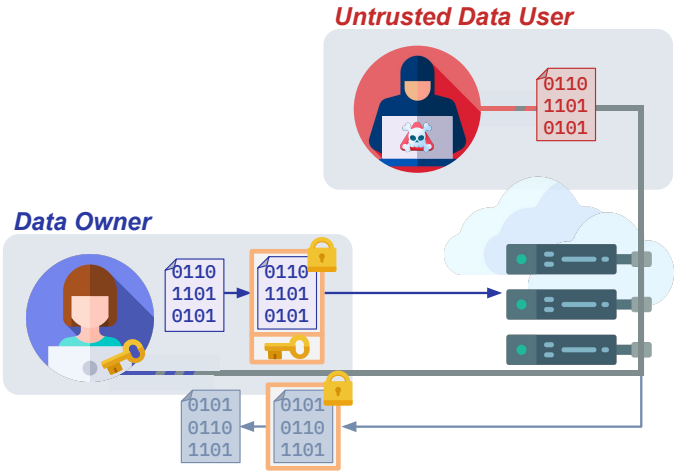


Figure 7.1: **Sequestered Encryption Usage Model.** We consider a data owner who wants to take advantage of an online service but does not trust this service with their data. The data owner can encrypt their data and access key with a standard encryption technology in a way that only grants access to the trusted hardware within the data user's server. Regardless of whether the data user is honorable, the owner's data remains confidential.

a willingness to place trust in hardware.[1]

In this work, we explore how trusted hardware can provide data confidentiality in the presence of vulnerable software while achieving practical performance overheads. We introduce ***Sequestered Encryption (SE)***—a hardware capability that computes directly on encrypted data in the microarchitecture while isolating related sensitive plaintext values from software. Specifically, SE ***sequesters*** sensitive plaintext data into a small trusted hardware unit (termed the SE unit) and, in turn, ***encrypts*** this sensitive data everywhere outside the SE unit, including in the register file and other microarchitecture structures. With this approach, ***no sensitive plaintext values exist in any software-accessible state***, significantly reducing the attack surface. Further, this approach drastically shrinks the trusted computing base compared to other hardware defenses since users only need to trust the small SE unit hardware, which lends itself to formal verification.

Use Cases. SE is designed to provide confidentiality for data processed in an untrusted or potentially vulnerable environment. Thus, SE is suited for cloud services processing third-party data on a public web server or another untrusted cloud-based platform, or in an environment co-located with untrusted processes. In these settings, third-party data can be

[1]Trusted execution environments (TEEs) do not guarantee data privacy in the presence of vulnerable software, as they typically trust some software components and exclude timing side channels from their treat models. These choices leave them vulnerable to attack [118, 219, 220, 221].

encrypted in a trusted, client-side environment and offloaded to the server for computation. This usage model is illustrated in Figure 7.1.

7.1.1 Chapter Contributions

We believe that architectural approaches can emerge as comprehensive data privacy solutions that possess zero software trust while being dynamic, expressive, and performant. In support of this argument, this chapter makes the following contributions:

- We present Sequestered Encryption (SE)—a hardware capability that provides comprehensive data privacy by removing all sensitive plaintext values from software-accessible architectural state.
- We demonstrate that our solution is expressive by evaluating a diverse set of 21 workloads in our analysis.
- We provide a detailed evaluation of the performance overheads of our designs. We find that encryption latencies have a near-linear impact on the overheads of SE.
- We demonstrate that our proposed SE design is resilient to cryptanalysis and timing side channels.

In Section 7.2, we introduce Sequestered Encryption (SE) and describe our security goals and threat model. In Section 7.3, we present the architecture and microarchitecture of SE. We then discuss the software interfaces in Section 7.4 before presenting our evaluation in Section 7.5 through Section 7.7. Finally, we discuss related works in Section 7.9 and conclude in Section 7.10.

7.2 Protecting Data In Use with Sequestered Encryption (SE)

Sequestered Encryption (SE) is a hardware capability that computes directly on encrypted data in the microarchitecture while isolating related sensitive plaintext values from software. Specifically, SE *sequesters* sensitive plaintext data into a small trusted hardware unit (termed the SE unit) and, in turn, *encrypts* this sensitive data everywhere outside the SE unit, including in the register file and other microarchitecture structures. With this approach, no sensitive plaintext values exist in *any* software-accessible architectural state, significantly

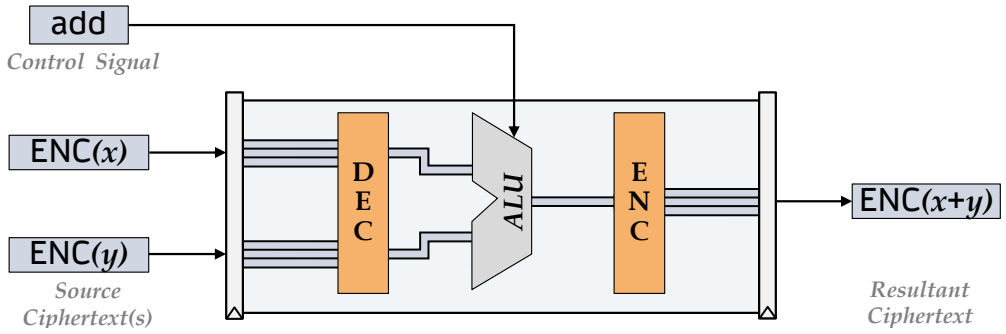


Figure 7.2: **Simple Design of Sequestered Encryption Functional Unit.** A simple implementation of SE is an arithmetic logic unit (ALU) fitted with cryptographic engines. Data decryption, computation, and re-encryption are done as one atomic operation between two latches. The computation is selected by a public control signal.

reducing the attack surface. We assume that the data encryption scheme ensures the confidentiality of ciphertext values stored outside of the SE unit.[2] Under this assumption, ciphertexts can persist in unverified or leaky hardware structures at no risk to data confidentiality. Decryption capabilities and keys are isolated within the SE unit, permitting the unit to process encrypted values while ensuring that the SE unit is the only hardware structure that can access sensitive plaintext values via decryption.

Our architecture design (§7.3) implements SE as a small hardware *functional unit* embedded within the pipeline. As a result of this design, operations can be dispatched to the SE unit on a per-instruction basis, enabling tight interleaving of secret and non-secret computation and minimizing overheads. Our microarchitecture design (§7.3.3) ensures that *i)* No sensitive plaintexts within the SE unit are exposed outside of it, and *ii)* Any operation (or sequence of operations) performed within the SE unit is free of timing side channels. Conceptually, the simplest design of the SE unit is an arithmetic logic unit (ALU) fitted with cryptographic engines, as shown in Figure 7.2. When instructions are dispatched to this unit, the input operands are decrypted, a single operation is computed (*e.g.,* addition), and the result is re-encrypted between two latches. While this design is uncomplicated, encryption latencies exist on the critical path, warranting performance optimizations. In this chapter, we present and evaluate a stateless atomic implementation of the SE Unit (Figure 7.2). In Chapter 8, we present a stateful optimized implementation that serves to reduce the exposed latencies of runtime encryption operations. Below, we discuss the security goals that guide our design and optimization of SE before discussing our implementation in Sections 7.3.

[2]The data encryption scheme is parameterizable and can be chosen to match the performance-security needs of the system.

7.2.1 Security Goals

Our goal is to maintain the confidentiality of a user's secret data in the presence of untrusted software. For clarity, we refer to this user as the ***data owner*** in the remainder of this text. In our intended use case, this data owner is the client offloading their data to a server for computation. Secret data is the plaintext value of any program variable explicitly denoted as "Private" and encrypted by the data owner. All other program data is assumed to be non-secret, including code contents and ciphertext values, as we assume that the adversary can easily infiltrate the software processing the data (§7.2.2). Specifically, we adopt the following definitions:

- ***Secret Data:*** Plaintext values of program variables denoted as Private
- ***Non-Secret Data:*** Ciphertext values of program variables denoted as Private; Plaintext values of program variables not denoted as Private; Constant values and instruction immediates; Instructions and instruction properties (*e.g.*, data dependencies, instruction addresses)

Our security goal aims to maintain the confidentiality of secret data on the server under two potential threats: ***secret data disclosure*** and ***secret-data-dependent timing***. To achieve this goal, we encrypt the representation of secret data values (I-9). The plaintext values of secret data can never be exposed outside of the SE unit or downgraded to non-secret. Only the data owner can decrypt and release their data after receiving a response from the server. To protect against secret-data-dependent timing, the timing of the SE unit must be independent of the plaintext values of secret data (I-11). Together, these goals prevent adversaries from directly learning secret data or inferring it via timing, control, or memory side channels.

7.2.2 Threat Model

We assume a software-based adversary that can exploit the vulnerabilities expected in systems software to gain full system access. In particular, we assume that an attacker can compromise any operating system or hypervisor present on the machine and launch malicious code in user or privileged mode. The attacker has complete knowledge of the machine's software, architecture, and microarchitecture, including the SE enclave, but does not have knowledge of secret key information. The attacker can passively analyze system data, directly access register and memory values, and mount active attacks (*e.g.,* install software to load and compute directly on secret data). The attacker can also exploit precise hardware timers,

speculative state, and shared microarchitectural state that is accessible via software-based attacks (*e.g.,* Spectre [16]).

Within the context of our usage model, this threat model considers a cloud service that is actively malicious, is compromised by a remote attacker, or is hosted by a malicious administrator or service provider. The goal of this adversary is to steal secret data that is present on the device.

Root of Trust. To be assured of the confidentiality of their secret data, the data owner must trust that the SE unit hardware is implemented in a secure and bug-free manner. The data owner must also trust attestation and key exchange with the SE hardware located on the server. However, the data owner does not need to trust any hardware outside of the SE unit (*e.g.*, speculation logic, memory, *etc.*) or any software running on the SE device, including the operating system.

SE is unique in that its trusted computing base (TCB) is small and contains *no* software. Additionally, trust in the SE hardware can readily be established as SE's compact design lends itself to formal design validation.

Non-Goals. We assume the adversary does not have physical access to the machine and therefore cannot leverage physical attacks, physical side channels (*e.g.*, power), or transistor-scale voltage measurement (*e.g.*, focused ion beam sensing). Additionally, we assume that the SE unit hardware is implemented correctly and is free of hardware Trojans. SE does not protect the integrity of code or memory since we assume our attacker has unlimited ability to manipulate computation. Finally, we exclude attacks that target data not denoted as private and not encrypted by the data owner.

7.2.3 Novelty of Proposed Approach

SE is a novel hardware technique characterized by its comprehensive use of encryption and minimal trusted computing base (TCB). Hardware defense techniques have mainly employed permissions checking (*e.g.*, Intel SGX [218]), physical isolation (*e.g.*, Ascend [222]), and isolation through the flushing and partitioning of shared structures (*e.g.*, MI6 [126]). Some hardware defenses do use encryption to protect program data, code, or pointers. However, these approaches lack comprehensiveness, as they decrypt values at the processor-cache boundary [27, 28], with many stopping further down the memory hierarchy [31, 218, 223]. For example, while Intel SGX does use memory encryption, it sits at the edge of the on-chip memory hierarchy and, therefore, is unable to protect enclave memory from software attacks [216].

SE's compact functional unit design results in a significant reduction in the overall TCB compared to other hardware defense mechanisms (§7.2.2). This minimal design lends itself to formal verification. As noted by prior work [216], formally reasoning about enclaves like Intel SGX is presumably infeasible as any proof would have to model all processor features that expose registers. Further, such work would be short-lived as any architectural modifications would invalidate this security proof. In contrast, SE's compact design is independent of many pipeline structures, avoiding these deficiencies.

While SE leverages similar defense mechanisms to the hardware techniques mentioned above, its security goals and threat model are significantly different. Namely, trusted execution environments (TEEs) and other secure enclaves are designed to maintain program integrity, often disregard side channels [120, 216, 221], and assume either a trusted operating system or application software [219, 220]. SE focuses solely on guaranteeing data confidentiality from both direct disclosures and timing side channels in a wholly untrusted software environment. For these reasons, the goals and threat model of SE are more closely related to homomorphic encryption (HE). HE has a smaller TCB than SE, since HE trusts only the cryptography itself. However, SE trades this trust in the SE hardware to build a solution that is more dynamic, expressive, and performant (§8.4.2), addressing many of the core issues that have inhibited HE's adoption [224, 225].

7.3 The Sequestered Encryption Architecture

Sequestered Encryption (SE) is a hardware capability that protects the confidentiality of secret data by sequestering all sensitive plaintexts into the trusted SE unit and encrypting these values in all other hardware structures. SE is architected as a small hardware functional unit embedded within the execute state of the pipeline. The interfaces to the SE unit are implemented as an instruction set extension such that all outputs are encrypted and all timing is independent of secret data, thereby thwarting attacks that attempt to leak or infer sensitive plaintext values. In this section, we discuss the architecture of the SE unit—its functionality and instruction-level interfaces—and our stateless microarchitecture implementation.

7.3.1 Hardware Architecture

We implement SE as a hardware functional unit embedded within the execute stage of the x86 pipeline, as shown in Figure 7.3. The SE unit is commanded via an extension to the x86-64 instruction set architecture (ISA), summarized in Table 7.1. When the processor decodes an instruction from the SE ISA extension, the instruction is dispatched to the SE

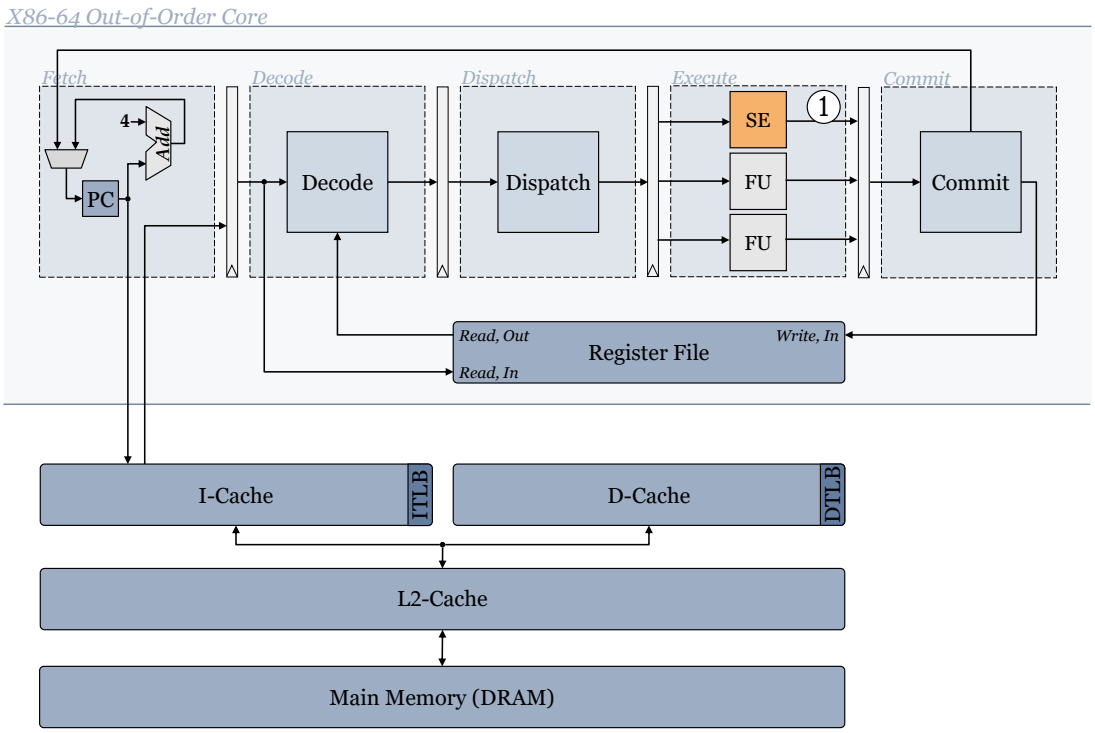


Figure 7.3: **Sequestered Encryption Architecture.** Sequestered Encryption (SE) is architected as a functional unit that processes instructions belonging to the SE ISA Extension (Table 7.1). Thus, the x86-64 core is extended with an SE functional unit that computes on encrypted source operands ①. In addition, the register file is extended to accommodate 128-bit encrypted data types. No other modifications to the memory system are made.

unit for processing. The instruction opcode indicates the format of the source operands (*e.g.*, ciphertext-plaintext) and the operation for the SE unit to perform (*e.g.*, addition). This process mirrors how native x86-64 instructions are dispatched to functional units in the execute stage of the pipeline.

The processor and its software perceive the SE unit as a black box that receives control signals and data inputs and produces a result. Like other pipeline functional units (*e.g.*, the ALU), software is unable to access any microarchitectural state within the SE unit but can read the SE unit's inputs and outputs and time its overall latency. By designing the SE microarchitecture to encrypt all outputs and have deterministic timing, we ensure that any software-observable values do not compromise the confidentiality of the secret data processed within the SE unit.

Instruction Class	Example	Semantics
Arithmetic	`enc_add x, y, z`	`x = enc(dec(y) + dec(z))`
Relational	`enc_sleq x, y, z`	`x = enc(dec(y) ≤ dec(z))`
Boolean	`enc_and x, y, z`	`x = enc(dec(y) & dec(z))`
Conditional Move	`enc_cmov x, y, z`	`x = enc(dec(y) ? dec(x) : dec(z))`
Memory Access	`enc_mov [x] →y`	`mov [x]→y; mov [x+0x8]→y`

Table 7.1: **Sequestered Encryption ISA Extension.** The x86-64 ISA extension facilitates encrypted computation of arithmetic, relational, and Boolean operators and conditional moves within the SE unit. Loads and stores of encrypted values (`enc_mov`) are decomposed into two 64-bit `mov` micro-ops, as discussed in Section 7.3.1.2.

7.3.1.1 Source Operands

The SE unit accepts source operands in multiple formats, as the SE ISA extension supports ciphertext-ciphertext, ciphertext-plaintext, and ciphertext-immediate inputs. Ciphertext inputs are expected to have the format illustrated in Figure 7.4. Namely, each ciphertext is constructed from an 8B secret data word appended with a unique 8B random cryptographic salt, which serves to diversify the ciphertext and thwart cryptanalysis attacks. The resultant 16B (128-bit) value is then encrypted under the symmetric encryption scheme supported by the SE unit. Any secret data values that are shorter in length (*e.g.*, 4B integers) are padded accordingly. When the SE unit receives a ciphertext source operand, the value is decrypted (inversely to Figure 7.4) and the random salt is discarded. The computation proceeds on the plaintext secret data word. The result is then formatted and encrypted using a fresh salt value before being emitted from the SE unit.

The data encryption scheme is parameterizable and can be chosen to match the performance-security needs of the system. In this work, we evaluate several encryption schemes that represent varying latencies and security: AES-128 [10], Simon-128/128 [13], and QARMA$_{11}$-128-σ_1 [10]. Data owners must prepare their sensitive inputs to match this data encryption scheme. Software support for writing programs within SE is discussed in Section 7.4.

7.3.1.2 Control Interface (ISA Extension)

The processor explicitly commands the SE unit via its defined control interface—the x86-64 ISA extension summarized in Table 7.1. The control signals provided to the SE unit are non-secret instruction opcodes that indicate the format of the source operands and the operation to compute. Below, we discuss the instruction classes that constitute the SE ISA extension.

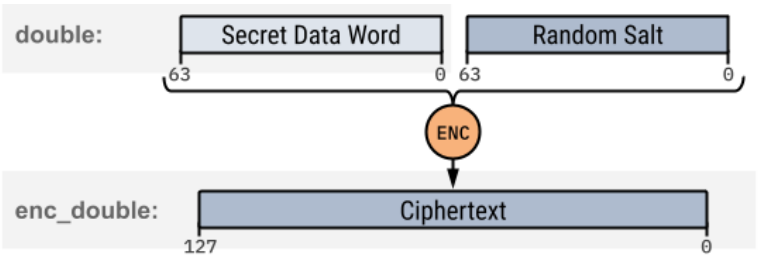


Figure 7.4: **Encrypted Data Layout.** Primitive types (*e.g.*, `double`) are appended with an 8B random cryptographic salt and then encrypted to form a 128-bit encrypted data type (*e.g.*, `enc_double`).

Loading and Storing Encrypted Data (MOV): Encrypted data is loaded from memory for processing via an `enc_mov` instruction. This instruction is handled by the main processor's memory interface. This operation is modified slightly because the encrypted data is 128-bits long, whereas our modeled memory system accommodates 64-bit data paths. Rather than modifying the processor's data paths, we decode the `enc_mov` instruction into two 64-bit `mov` micro-ops to jointly load the entire 128-bit value into the register file. The register file is extended to 128-bits to accommodate these extended ciphertexts. Encrypted store instructions proceed equivalently. The operation of x86-64 load/store instructions is unaffected by these changes, as the lower 64-bits of the register file can be accessed independently and the data path is unmodified. These `enc_mov` commands do not decrypt data, as sensitive data in the register file always remains encrypted.

Operating on Encrypted Data (ADD, MUL, *etc.*): Once ciphertexts are loaded into registers, they are ready for processing by the SE instructions. The SE unit supports arithmetic (integer and floating-point), Boolean, and relational instructions, as listed in Table 7.1. Each instruction must have deterministic (data-independent) timing to uphold our security goals. Thus, we prohibit hardware optimizations within the SE unit that accelerate instructions for specific inputs (*e.g.*, early termination for multiply-by-zero [226]), as these optimizations leak information about sensitive plaintext values via timing. Different SE instructions still have different latencies within the SE unit (*e.g.*, `enc_add` takes one cycle, whereas `enc_mul` takes three cycles), as these timing differences depend solely on non-secret instruction opcodes rather than secret data inputs.

Making Decisions on Encrypted Data (CMOV): The SE architecture supports conditional move (`enc_cmov`) operations, akin to the x86-64 `CMOVcc` primitive. Conditional moves function as ternary operators, where a destination register is updated depending on the value of a secret condition. This primitive enables programs to make decisions on en-

crypted conditions without leaking information via timing or exposed state, as described further in Section 7.4.

7.3.2 A Subtractive Approach to Enforcing our Security Goals

The SE ISA extension and its semantics are designed such that it only supports *oblivious computation*. Also called "constant time" or "data-oblivious" programming, this technique avoids side channels by ensuring that program execution is independent of secret data [109, 227, 228, 229], thereby enforcing our security goals (§7.2.1).

In SE, we achieve oblivious computation by *i)* Isolating decryption capabilities and keys within the SE unit, and *ii)* Only implementing oblivious operations within the SE unit. Traditionally, oblivious computation is difficult to achieve because many non-oblivious instructions exist in the native x86-64 instruction set (*e.g.*, conditional branches). However, the functional units that compute these native instructions are unable to access secret data via decryption since decryption capabilities are isolated within the SE unit. Thus, any native, non-oblivious instructions are performed on *ciphertext values*, producing garbled results and leaking no sensitive information. The SE unit, on the other hand, is capable of decrypting values and accessing sensitive plaintexts. Thus, to enforce oblivious computation, we *only* implement oblivious operations within the SE unit—arithmetic, relational, and Boolean instructions and conditional moves. The SE hardware simply *cannot* process conditional branches or instructions accessing encrypted memory addresses.[3] Together, this design forces oblivious computation on secret data at the hardware level.

We discuss the program transformations required to support oblivious computation in Section 7.4. Programs can still perform non-oblivious operations on non-secret data, reducing the required program transformations and improving performance. Finally, we note that oblivious computation is required for *all* secret computation frameworks, including HE, as secret-data-independent control flow and program execution is fundamental to preserving confidentiality.

7.3.3 SE Unit Microarchitecture

As presented in Figure 7.2, our simplest design of the SE unit is a functional unit bookended with cryptographic engines. The cryptographic engines and computational hardware sit between two latches, which provide the entry and exit point for the SE unit data path.

[3] Alternatively, one could permit encrypted address operands by using Oblivious RAM (ORAM) [105] to obfuscate memory access patterns.

We outfit the SE functional unit with an arithmetic logic unit (ALU) and a floating-point unit (FPU) to implement the semantics of the SE ISA extension. Furthermore, we pipeline both the cryptographic engines and computational hardware to improve throughput and reduce performance impacts on overall runtime. To uphold our security goals, we design our microarchitecture such that *i)* No secret plaintexts within the SE unit are exposed outside of it, and *ii)* All computation within the SE unit has deterministic timing (*e.g.*, there are no optimized paths for multiply).

The processor pipeline is augmented to dispatch instructions to the SE unit, as done for any other functional unit cluster in the execute stage. Namely, the processor sends the source operands and control signals (*i.e.*, instruction opcode) to the SE unit once available. This logic is identical to dispatching instructions to any other functional unit cluster in the execute stage.

The SE unit can be extended to meet the computational requirements of the native processor. For example, if the processor supports single-instruction multiple-data (SIMD) operations, the SE unit could be built to include SIMD computational capabilities. Additionally, the SE unit could include multiple cipher engines or computational units to further increase throughput (*i.e.*, a superscalar implementation). In this chapter, we evaluate an SE unit with a single instance of each component, which we regard as the most conservative design.

7.3.3.1 Walkthrough

The stateless microarchitecture is shown in Figure 7.2. When a ciphertext value enters the SE unit, it is fed to the decryptor, which decrypts the source operands and discards the salt. Then, the plaintext values are forwarded to the computation hardware, which operates on them as indicated by the public control signal (*e.g.,* `enc_add`). When the computation completes, the encryptor re-encrypts the result with a new salt value. This final ciphertext is fed to the latch exiting the SE unit. This path through the encryptor and corresponding latch is the only exit point from the SE unit.

7.3.4 Key Infrastructure

Prior work has refined hardware-based key management to the point where we can integrate these techniques with few modifications (*e.g.*, [230, 231]). To facilitate communication between the data owner and the server-side SE architecture, the SE unit is extended to support a key exchange protocol that establishes a shared data encryption key (k). The SE unit is also defined to contain dedicated, isolated hardware registers to store this key (k) and those

used for key exchange (pk_{se}, sk_{se}). These registers are not accessible to software-based addressing and are only read/written implicitly though the SE interface (*e.g.*, `expose_key`). The SE unit's private key (sk_{se}) can be generated in hardware using a memory physically unclonable function (PUF), as done in [232, 233]. Hardware-supported attestation [216, 218] and key exchange allow the data owner to authenticate the SE unit and agree on a shared data key (k). Namely, to perform key exchange using RSA:

1. The data owner acquires the public key (pk_{se}) of the SE hardware from a trusted certificate authority.

2. The data owner generates a symmetric key (k) that will be used as the shared data encryption key. They encrypt k with pk_{se} according to the RSA protocol.

3. The encrypted key packet and encrypted data is transported to the server containing the SE extension.

4. The server OS receives the packet and issues an `expose_key` instruction.

5. `expose_key` is dispatched to the SE unit, which decrypts the key packet using the SE unit's private key, sk_{se}, and stores the result in the isolated key register.

6. Now, protected computation using SE can proceed with the data owner's symmetric key, k.

The semantics of `expose_key` are implemented within the SE unit to uphold the SE security assumptions and have deterministic timing. For RSA, this requires that the exponentiation implementation does not optimize for zero key bits.

7.3.5 Process-OS Interface

Applications using SE must still run atop and interact with the operating system (OS) for routine tasks, like context switching and exception handling. SE's key mechanisms enable programs to make these requests without having to trust this complex (and likely vulnerable) systems software to protect data privacy. On a context switch, the OS must swap in the data encryption key of the new context. To stash the existing data encryption key (k), the OS issues a `conceal_key` instruction, which is the functional inverse of `expose_key`. Specifically, this instruction encrypts k with the SE unit's public key, pk_{se}, and writes the resultant packet to a register. This key packet can then be stashed into storage, permitting multiple contexts to reside at once. We note that this key packet does not pose an additional security risk, since it has already been transmitted during key-exchange. The OS, if uncompromised, will use the

`expose_key` and `conceal_key` instructions to perform context switches appropriately. If the OS is malicious, data corruption will occur from key mismanagement, affecting correctness but not violating our confidentiality goals (§7.2.1).

To prevent exceptions from revealing secret data, exception indications must be embedded inside the encrypted result. For example, a divide-by-zero exception cannot be revealed; otherwise, it would be possible to determine that the confidential denominator is zero. As done in prior work [109, 227], the SE unit masks such exceptions by replacing the plaintext with a canonical value indicating the fault. Subsequent operations propagate this value, thereby allowing the fault to be discovered when the final result is decrypted by the data owner.

7.4 Programming with Sequestered Encryption

Sequestered Encryption (SE) extends the processor with the capability to recognize secret data and selectively protect the confidentiality of these values. Applications can engage the SE instruction interface to extend these privacy guarantees to their users. To provide access to these data security features, we extend programming environments to support SE semantics, as detailed in this section.

7.4.1 Abstractions for High-Level Languages

We abstract the SE interfaces in a C++ library, enabling programmers to easily engage SE's protections for particular program variables. Our library implements 128-bit encrypted data types (*e.g.*, `enc_int`) and uses overloaded operators to emit SE ISA instructions (*e.g.*, `enc_add`) directly via inline *asm()* statements. This implementation encapsulates the SE ISA extension, allowing programs to compute with SE as they compute with primitive types (*e.g.*, `int`), shown in Figure 7.5. We chose this approach because it enabled fast prototyping. However, one could integrate this support into the compiler using static data-flow analysis, which would be better suited for a high-performance commercial implementation.

SE uniquely allows for tight coupling of secret and non-secret computation. Our library supports sensitive variables for all primitive C++ types, from 1B (*e.g.*, `enc_char`) to 8B (*e.g.*, `enc_double`). These types can be combined with non-secret types in structs or classes. Additionally, since instructions are dispatched to the SE unit on a per-cycle basis, public and private data types and computation can be tightly interleaved. Thus, algorithms can opt to use sensitive types to protect the most critical portions of computation. This reduces overheads for real-world applications by reserving encrypted operations for a minimal subset

```
1  void bubbleSort(enc_int *arr, int n){
2     for(int i=0;  i<n-1;  i++) {
3       for(int j=0;  j<n-i-1;  j++) {
4          enc_int  tmp  = arr[j];
5          enc_bool gt   = arr[j] > arr[j+1];
6          arr[j]   = enc_cmov(gt, arr[j+1], tmp);
7          arr[j+1] = enc_cmov(gt, tmp, arr[j+1]);
8       }
9     }
10 }
```

Figure 7.5: **Sorting a List with Sequestered Encryption.** This program sorts a list of encrypted integers using SE operations, facilitated by our GNU C++ library. Due to the restricted SE environment (§7.4.2), a conditional move is used in place of an if-statement.

of values. For example, the Wagner–Fischer algorithm is a well-known dynamic programming algorithm to calculate the edit distance between two substrings. Because the algorithm's looping structure and constants are public, only a small subset of instructions that interact directly with user data need to be encrypted (33%, as found in our *Edit-Distance* benchmark).

7.4.2 Programming Limitations

As discussed in Section 7.3.2, the SE unit only supports oblivious instructions. Despite this limitation, SE's conditional move primitive enables programs to still express secret-dependent conditional logic and memory addressing by leveraging the following transformations, which concede performance. We view compilers as promising mechanisms to assist in these code transformations [96, 227, 234, 235]. In Chapter 9, we discuss data-oblivious programming in more depth and evaluate the sole overheads of these transformations.

7.4.2.1 Secret-Dependent Conditional Logic

Control decisions on secret data can only be expressed using SE conditional move instructions as predicates. As a result, programs cannot use sensitive-data-dependent heuristics, early-exit conditions, or recursion, causing algorithms like *Bubble-Sort* to have worst-case time complexity. The following code snippet demonstrates how programs can compute an if-statement that increments `x` or `y` depending on an encrypted condition `secret`.

```
1  x = enc_cmov( secret, x+1, x);
2  y = enc_cmov(!secret, y+1, y);
```

7.4.2.2 Secret-Dependent Memory Addressing

Secret-based memory addressing or array indexing can be achieved (at a cost) by accessing every element and using a conditional move to select the desired value. Alternatively, SE could be adapted to use techniques like Oblivious RAM (ORAM) [105] to enable secure secret-based addressing, thereby reducing overheads related to these code transformations [109, 222]. The following code snippet demonstrates how programs can compute the secret-dependent memory access `arr[secret]`.

```
1  for(int i=0;  i<len(arr);  i++;)
2      ret = enc_cmov(i==secret, arr[i], ret);
```

7.5 Experimental Methodology

We implemented SE on a x86-64 out-of-order core in gem5 [6] in system call emulation mode with a 2.5 GHz clock frequency. We extend the core with: *i)* An SE functional unit in the execute stage (§7.3.3), *ii)* Extensions to the x86-64 ISA that dispatch the SE ISA instructions to the SE unit (§7.3.1.2), and *iii)* An extended 128-bit register file (§7.3.1.2). We analyze the stateless SE unit microarchitecture, denoted **SE**. Below, we detail the encryption schemes and workloads used for our analysis.

Core Type	2.5GHz, O3CPU (Out-of-Order)
Superscalar	4-wide
Cache Line Size	64B
L1 Cache	32KB with 2-cycle latency
L2 Unified Cache	256KB with 20-cycle latency

Table 7.2: **Experimental Configuration for gem5.** The specifications of our baseline x86-64 out-of-order core in gem5 [6].

Evaluated Data Encryption Schemes. The data encryption scheme used for SE is parameterizable and can be chosen to match the performance-security needs of the system. In this work, we evaluate three symmetric encryption schemes: *i)* AES-128 (40 cycles), *ii)* Simon-128/128 (20 cycles), and *iii)* QARMA$_{11}$-128-σ_1 (12 cycles). The cipher latencies were derived from hardware implementations reported by prior work [10, 13] and scaled to a 7 nm technology node and 2.5 GHz clock using the techniques in [11]. We use a Mersenne Twister pseudorandom number generator (PRNG) [236] to generate cryptographic salts, re-seeded with a true random source every 200 uses. We used existing open-source

encryption libraries within our simulation [237, 238, 239] and parameterized gem5 with the above latencies.

Compiler Support and Benchmarks. We evaluated SE on 21 benchmarks from the VIP-Bench privacy-focused benchmark suite compiled using GNU g++ with -O2 optimizations [15]. VIP-Bench annotates secret variables in its native workloads, enabling us to easily port all annotated VIP-Bench variables to use the encrypted types defined by SE's C++ library (§7.4). Furthermore, VIP-Bench includes both native and data-oblivious variants of each workload. This feature enables us to evaluate SE's overheads compared to a native (unprotected, non-data-oblivious) baseline, as well as analyze the software overheads associated with data-oblivious programming transformations.

7.6 Experimental Results and Evaluation

We assess the performance overheads of SE against a native baseline. Figure 7.8 shows the results for SE configured with QARMA, Simon, and AES across our evaluated workloads. Below, we detail the instruction and performance overheads of our design.

7.6.1 Overheads of Program Modifications

Workloads running on SE may experience an increased dynamic instruction count due to required program transformations. We report these overheads in Figure 7.6. We find that, in many cases, overheads related to data-oblivious modifications (shown in blue) are minimal, as computational benchmarks like *Edit-Distance* are oblivious by default. Data-oblivious modifications for *Bubble-Sort* actually result in 7% fewer committed instructions because *i)* An if-statement is replaced by a conditional move, and *ii)* A secret-dependent heuristic is removed. Both of these changes occur in the main body of the *Bubble-Sort* algorithm, amplifying their impact on the dynamic instruction count. A few benchmarks exhibit more than a 50% increase in dynamic instructions due to data-oblivious code transformations (*i.e.*, *Distinctness*, *Flood-Fill*, and *Kepler-Calc*).

Dynamic instruction count also increases due to use of the SE C++ library (shown in orange), which emits additional loads/stores for encrypted data types (§7.4.1). Workloads experienced an average 143% increase in load instructions due to the use of 128-bit encrypted data types for secret values. We believe that these loads, in addition cipher latencies, are the main source of overhead.

The relative *proportion* of dynamic instructions dispatched to the SE unit can provide

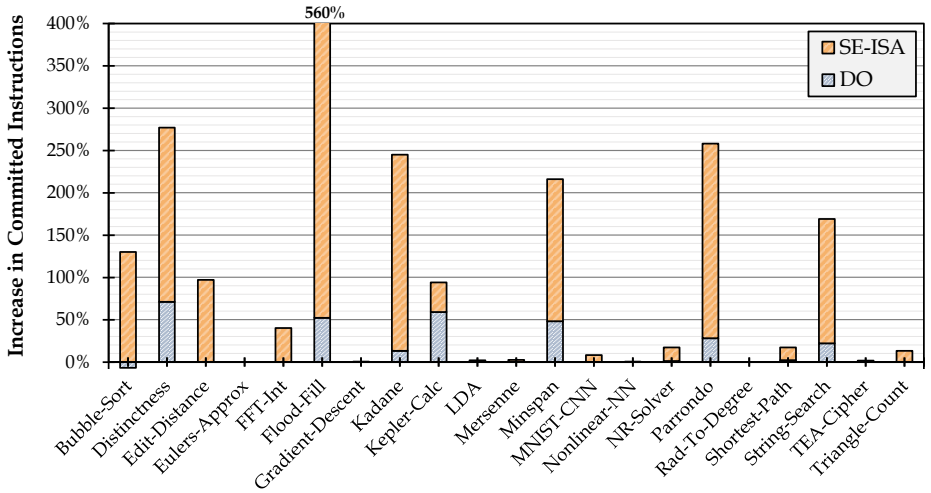


Figure 7.6: **Increase in Dynamic Instruction Count.** This figure shows the increase in dynamic instruction count due to data-oblivious program modifications (DO) and usage of the SE GNU C++ library, including additional loads for 128-bit data types (SE-ISA).

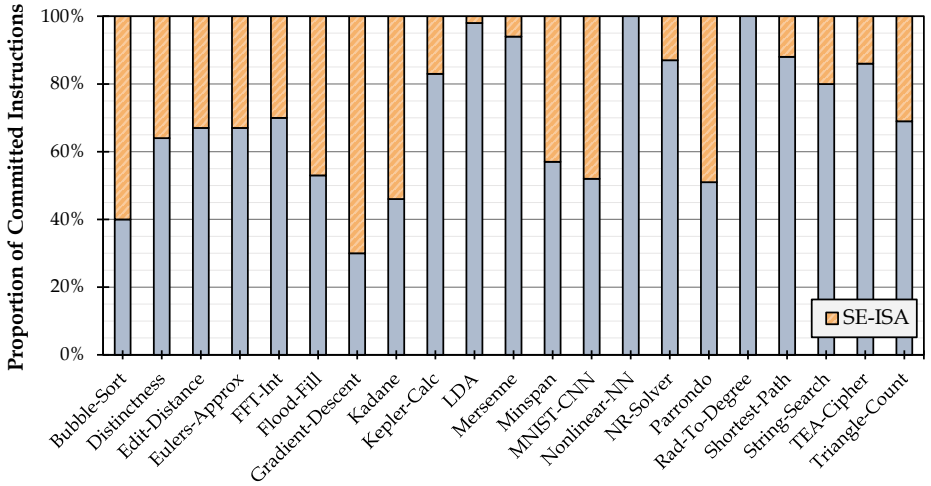


Figure 7.7: **Proportion of Dynamic Instruction Count.** This figure shows the proportion of dynamic instructions dispatched to the SE unit (*i.e.*, belonging to the SE ISA extension). On average, 30% of committed instructions were from the SE ISA extension.

intuition to the performance overheads caused by exposed cipher latencies. This analysis is shown in Figure 7.7. Such effects are not necessarily captured by Figure 7.6, as changing an x86-64 `add` to an SE `enc_add` does not result in an increase in the overall instruction count but is noticeable when studying the proportion of instructions. *Eulers-Approx* illustrates this case, as it experiences a negligible increase in instructions, yet 33% of its committed instructions are of the SE ISA. Across all workloads, 30% of committed instructions were dispatched to the SE unit. Some benchmarks only exhibit a small proportion of SE instructions (*i.e.*, *LDA*, *Nonlinear-NN*, *Rad-To-Degree*) because I/O logic overshadows the private computation contained in the benchmark.

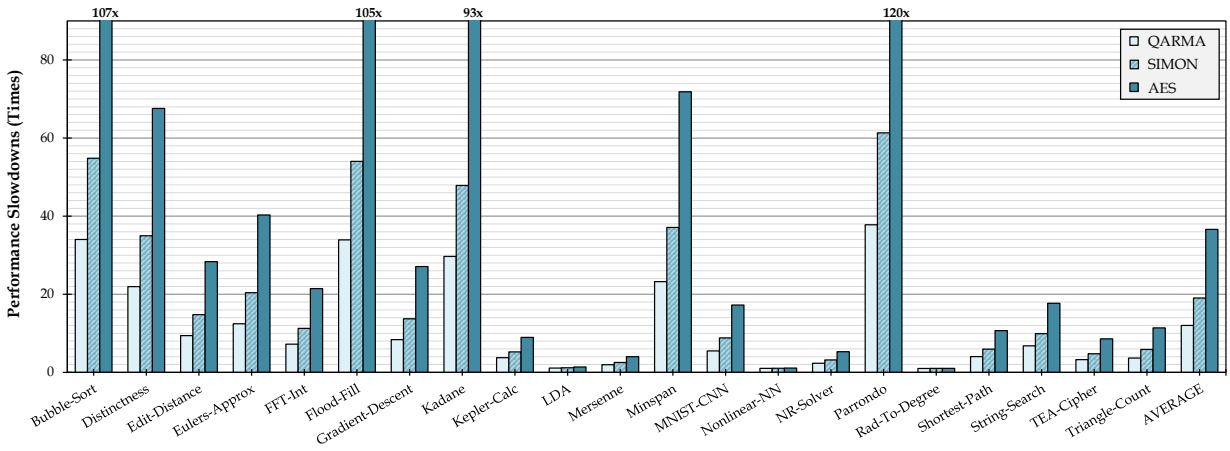


Figure 7.8: **Performance Slowdowns for Sequestered Encryption.** This figure shows the overall performance slowdowns for the stateless SE microarchitecture (SE) when parameterized with 12-cycle QARMA, 20-cycle Simon, and 40-cycle AES encryption. Performance overheads across different ciphers scale almost linearly with the cipher's latency.

7.6.2 Runtime Performance Overheads of SE

Performance slowdowns incurred by our stateless microarchitecture design (SE) are significant at 36.6×, 19×, and 12× for AES-128, Simon, and QARMA, respectively. These overheads reaffirm fears of adding long-latency cipher operations into computation streams. However, we see opportunities for improvement. Further, these overheads are far less than that of competing data privacy mechanisms.

Our observed slowdowns have a near-linear relationship to the latencies of the chosen encryption scheme. For example, *Edit-Distance* has a 14.7× performance degradation with the 20-cycle Simon cipher. When upgrading to the 40-cycle AES-128 cipher, slowdowns nearly double to 28.4×. Given this result, lightweight ciphers have the potential to greatly reduce the overheads of SE. For example, a hypothetical cipher requiring 3 cycles for an encryption operation would have a 4.2× performance slowdown, on average, across our analyzed workloads. Such low-latency ciphers are in development and could benefit SE in the near future [240]. In Chapter 8, we explore ways to mitigate performance overheads without compromising the encryption scheme.

Trends and Takeaways. Performance overheads incurred by our stateless microarchitecture, SE, are significant at 12-37× for our analyzed encryption schemes. Lightweight ciphers have the potential to significantly reduce these slowdowns as we've observed a near-linear relationship between the latency of the encryption scheme and overall performance overheads.

7.7 Security Analysis

Sequestered encryption wrestles trust from software by performing all sensitive computation behind a wall of encryption with support from a physically isolated and trusted hardware extensions. With adherence to this design, concerns about data privacy shift toward the timing characteristics of the SE computation and the ciphertexts produced by those computations.

In this section, we discuss how the ciphertexts produced by the SE unit are resistant to cryptanalysis and information leakage. The ciphertexts of secret values are exposed outside of the SE unit in software-accessible or leaky hardware structures (*e.g.*, the register file). We assume that the chosen data encryption scheme ensures confidentiality of these values outside of the SE unit. This assumption can readily be met via the use of strong ciphers like AES-128, Simon-128/128, and QARMA$_{11}$-128.

7.7.1 Qualitative Discussion of Security of Evaluated Encryption Schemes

Of our assessed encryption schemes, AES-128 is the most tested and secure, as it has been subject to more than twenty years of unsuccessful cryptanalysis.[4] Emerging lightweight block ciphers, like Simon-128/128 and QARMA$_{11}$-128, can reduce exposed pipeline latencies while still providing adequate security. Simon-128/128 is a block cipher with a 128-bit key that has been optimized for hardware [200, 201]. Both Simon-128/128 and QARMA$_{11}$-128 have demonstrated significant resistance to cryptanalysis and provide 128-bits and 256-bits of security, respectively [10, 201]. Simon has been subject to some analysis; the best known attacks are via differential cryptanalysis, which is only marginally faster than brute-force attempts and therefore not practical. QARMA$_{11}$-128 has 256-bits of security [10] and executes more than three times faster than AES. QARMA is more amenable to algebraic cryptanalysis attacks compared to AES. However, the only known attacks on QARMA having significant time complexity [241]. While these two algorithms have not received the level of evaluation as AES-128, they have demonstrated significant resistance to cryptanalysis.

7.7.2 Resilience to Distinguishing Attacks

Even with the use of strong ciphers, our data encryption scheme must use cryptographic salts to prevent attackers from making correlations between ciphertexts throughout computation,

[4]While implementation attacks have been demonstrated [97, 98, 99], we avoid these pitfalls with a side-channel resistant hardware design.

as correlating the input and resultant ciphertexts for a conditional move instruction trivially reveals the secret condition. In this section, we evaluate the resilience of our salting scheme to cryptanalysis.

We demonstrate that our data encryption scheme is resilient to *distinguishing attacks*—a primary form of cryptanalysis where an attacker can distinguish ciphertexts from purely random data due to the presence of patterns, allowing them to recover sensitive values from relations between inputs and outputs. We use the Dieharder [242] random number test suite to analyze the similarity of ciphertexts and detect correlations between sensitive plaintexts and main-core-observed data traces. Dieharder includes statistical tests from the NIST standard for random and pseudorandom number generators [243]. We ran Dieharder on encrypted inputs, outputs, and intermediate values produced by our genomic *Edit-Distance* benchmark, which is of particular interest because it contains only 13 unique data values (*i.e.*, four nucleotides and nine integer/Boolean values). To demonstrate that the data set contains significant correlations, we ran Dieharder when using only weak XOR encryption. All tests failed, indicating the significant patterns in the benchmark's data traces. When securing *Edit-Distance* using SE's encryption scheme under AES-128, Simon, and QARMA, ***all 114 Dieharder randomness tests passed***, demonstrating that our design presents a significant challenge to cryptanalysis attacks.

7.8 Discussion and Limitations

The SE microarchitecture design proposed in this chapter stores no state, has a minimal trusted-computing base, and occupies a small hardware area. This uncomplicated design has the potential to simplify formal verification efforts, enabling us to establish trust in the SE implementation. However, the performance overheads of this design may be prohibitive, especially when using long-latency ciphers (*e.g.*, AES-128 has 36.6× slowdown on average). While this overhead is smaller than HE approaches, it still may be too large for protecting commercial applications. In Chapter 8, we discuss ways to reduce exposed cipher latencies with hardware optimizations. Another source of overhead is the use of 128-bit data types for encrypted values. In our implementation of SE, we model the memory system with 64-bit data paths, meaning that each `enc_mov` instruction dispatches two 64-bit load micro-ops. This additional load instruction accounts for a portion of our performance overhead. An alternative design could avoid this overhead by expanding the data paths to 128 bits.

A limitation of SE is that, given its implementation in hardware, it is immutable. While we evaluate multiple cryptographic ciphers in this chapter, SE likely will only support one cipher in a commercial implementation. Thus, this defense is susceptible to attacks on

the cryptographic scheme discovered after manufacturing, as the hardware implementation cannot be "patched" easily.

7.9 Related Work

SE leverages trusted hardware to provide data confidentiality in the presence of compromised or untrusted software. SE's security goals are closest to homomorphic encryption (HE) [29]. However, SE's design methodology and implementation are similar to trusted execution environments (TEEs) [216, 218, 223]. Thus, we discuss both of these related works in this section. For direct comparisons between SE and related works, see Sections 7.2.3 and 8.4.2.

7.9.1 Homomorphic Encryption (HE)

Homomorphic encryption (HE) [29] is a class of encryption schemes that preserve specific operations performed on ciphertext values after decryption. HE provides strong data confidentiality while eliminating all trust in software and hardware—a data owner only needs to trust the mathematical foundations of the encryption scheme. However, currently available HE libraries suffer from high performance overheads and programming limitations. As mentioned in Section 8.4.2, bootstrapping is a costly HE operation needed to eliminate noise that builds over subsequent multiply operations. Bootstrapping is regarded as prohibitively expensive and, as a result, many libraries and applications simply limit multiplicative depth in order to avoid bootstrapping. In addition to this restriction, many libraries cannot readily express conditional move operators or non-linear functions (*e.g.*, SEAL [244]), making them far less expressive than SE. Emerging Boolean HE schemes like TFHE [245] theoretically can support conditional move operations. But, these schemes suffer additional slowdowns and have not seen significant adoption.

Current advancements in HE are focused on overcoming the prohibitive performance overheads of software libraries through the use of optimized algorithms [14, 246] and custom hardware [247, 248, 249, 250]. HEAX [248] is a high-performance architecture for CKKS homomorphic encryption that is able to achieve a 200× performance improvement for ciphertext multiplications. More recent work leverages SIMD processing and massively parallel architectures to gain significant speedups over HE software implementations [249, 250]. Unfortunately, emerging HE research rarely reports absolute performance numbers, motivating our direct performance comparison in Section 8.4.2. Commensurate comparisons are also challenging because the community does not use standard benchmarks and often reports performance overheads over a non-native baseline (*e.g.*, [248]).

7.9.2 Privacy-Preserving ISA Extensions

Data Oblivious ISA Extension (OISA) [109] presents a RISC-V ISA extension to thwart side channels by preserving data-oblivious execution in the processor architecture. Specifically, OISA uses dynamic information flow tracking to detect when sensitive values are used for an unsafe (non-oblivious) instruction. OISA does not encrypt sensitive values, leaving them vulnerable to other disclosure mechanisms. Encrypt-Everything-Everywhere (E^3) [251] presents a software framework and (optional) ISA extension that keeps sensitive data encrypted throughout computation by leveraging either homomorphic operations via its HE interfaces or non-homomorphic operations via an isolated hardware co-processor. While E^3 is conceptually similar to SE, its design is significantly different, as it uses larger ciphertext values (up to 4069 bits), has a much larger hardware footprint, and leverages HE schemes in its implementation.

7.9.3 Trusted Execution Environments (TEEs)

Trusted execution environments (TEEs) have significantly different security goals as they are designed to protect the integrity and confidentiality of program execution [222, 252, 253, 254, 255, 256]. While these designs provide some data confidentiality guarantees, they trust that the application software is honorable and often exclude timing side channels from their threat model.[5] For these reasons, we regard SE and TEEs as significantly different security mechanisms that employ similar hardware techniques. ARM TrustZone [223, 257] and Intel SGX [130, 216, 218, 258] isolate applications to enforce the confidentiality and integrity of programs with moderate overheads (<1.2× and 1.7-4.5×, respectively) [218, 259]. However, these secure processor extensions leave the L1-cache and register file unencrypted [216, 218], or fail to provide protection in main memory [223], leaving sensitive data vulnerable to exfiltration [118, 219, 220, 221, 254, 255].

Recent advances in TEEs have sought to eliminate microarchitectural side channels by further isolating sensitive processes. Sanctum [260] and MI6 [126] protect in-order and out-of-order processors, respectively, by flushing microarchitectural state and partitioning the LLC. Both achieve low performance overheads (1.1-1.2× for most workloads) with no programming limitations and trust enclave software within their TCB. The Ascend processor [222, 256] implements fixed-time execution and ORAM to prevent off-chip adversaries from learning the execution traces of untrusted software. This design provides strong security with moderate performance overheads (4-13×) but requires critical computation to

[5] Ascend [222, 256] is an exception to this statement as it regularizes function timings.

be outsourced to the Ascend chip, which is separate from the main processor. These designs have much larger trusted computing bases than SE, as comprehensive isolation and permissions checking requires extensive processor modifications.

7.10 Chapter Summary

Software systems are often trusted to store and process sensitive third-party data despite the pervasiveness of software vulnerabilities. In this work, we explore how a small hardware root of trust can provide confidentiality of third-party data in an untrusted or vulnerable environment. We introduce Sequestered Encryption (SE)—a hardware capability that provides comprehensive data privacy by removing all sensitive plaintext values from software-accessible architectural state. SE encrypts secret data in all external processor architecture and microarchitecture, ensuring that software does not have direct access to sensitive plaintext values. Our evaluation demonstrates that SE is expressive. In this work, we implement a diverse set of 21 workloads, including those that cannot currently be expressed by homomorphic encryption. Further, SE achieves 36× performance slowdowns over native execution and demonstrates opportunities for further optimizations.

CHAPTER 8

Optimizing Encrypted Data Architectures

Sequestered Encryption (SE) is a powerful encrypted architecture that protects private data with a minimally invasive design. However, while this design is uncomplicated, cryptographic operations on the critical path significantly increase the overhead of SE instructions. For example, with AES's 40-cycle cipher latency, an add instruction takes 81 cycles to compute in SE. In this chapter, we iterate on the stateless design to reduce performance overheads and bypass idempotent cryptographic operations on the critical path. With our proposed architectural optimizations, we show that SE is able to achieve performance that is within 2.5× of native execution when using AES-128 cipher engines, with 73% of this overhead due to cryptographic latencies and the remainder (81%) due to software overheads (*e.g.*, data-oblivious programming, 128-bit data loads). Lightweight ciphers, like QARMA, have the potential to further reduce overheads to less than 2× of native execution. Comparisons to HE platforms underscore the value of hardware-based secure computation. We show SE to be faster than Microsoft SEAL [244] by three orders of magnitude over the same workloads. With this work, we demonstrate that SE can be an extremely performant and expressive alternative to homomorphic encryption.

The research presented in this chapter appears in:

[35] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Fitsum Assamnew Andargie, Todd Austin. ***"Sequestered Encryption: A Hardware Technique for Comprehensive Data Privacy,"*** 2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED), 2022. DOI: 10.1109/SEED55351.2022.00014.

[261] Qinhan Tan, Shibo Chen, Yonathan Fisseha, Lauren Biernacki, Jean-Baptiste Jeannin, Sharad Malik, Todd Austin. ***"Security Verification of Low-Trust Architectures"*** ACM Conference on Computer and Communications Security (CCS), 2023. *To Appear*

8.1 Introduction

Sequestered Encryption (SE) applies encrypted architectures to protect private data in a wholly vulnerable or untrusted software environment. This approach mirrors that of vulnerability-agnostic design in Morpheus. Through encrypting data throughout all software-accessible microarchitectural structures, SE provides comprehensive data privacy guarantees. However, this protection comes at a cost. SE places cryptographic latencies on the critical path. For programs with tight data dependencies, these latencies significantly degrade performance. Our evaluation in Chapter 7 shows that SE incurs ∼36× performance slowdowns on average compared to native execution when using AES. While more efficient ciphers can reduce performance overheads, we see great potential for other architectural optimizations to mask encryption latencies without compromising security.

In this chapter, we propose two optimizations to the SE microarchitecture that take inspiration from classical architectural approaches like forwarding of values and caching for temporal locality. The *decryption cache* saves recently decrypted ciphertexts, accelerating values that are reused in subsequent instructions or stored then re-loaded from memory. The *register cache* forwards plaintext values around idempotent encrypt-decrypt operations that exist between true data dependencies. Together, these changes mask almost all cipher latencies within the SE unit without exposing plaintexts to the external microarchitecture.

With our proposed optimizations, SE's overheads decrease from 36× to just 2.5× over native execution when using AES-128 cipher engines. Of this overhead, more than half is attributed to software changes like data-oblivious programming, motivating future research in Chapter 9. For lightweight ciphers like QARMA, our optimizations mask cipher latencies almost entirely—in our evaluation, only 17.5% of overheads for optimized SE with QARMA are due to encryption delays. These architectural optimizations demonstrate that SE can emerge as a dynamic, expressive, and performant data privacy solution.

8.1.1 Chapter Contributions

In this chapter, we make the following contributions.

- We present architectural optimizations for Sequestered Encryption (SE-Opt) that reduce the cost of computing on always-encrypted data, and we detail the trade-offs associated with these optimizations.
- We provide a detailed evaluation of the performance overheads of our designs. We find that SE using AES-128 has 153% performance slowdowns (98% for QARMA), with

about 81% of these slowdowns due to software constraints and the remainder due to exposed cipher latencies.

- We compare the performance of SE to SEAL homomorphic encryption and find that SE offers a performance benefit by three orders of magnitude.
- We discuss how our proposed SE optimizations are resilient to timing side channels and are amenable to formal verification of these claims.

The remainder of this chapter is organized as follows. Section 8.2 revisits our security goals and details our optimizations to the SE microarchitecture. Sections 8.3 and 8.4 discuss our experimental methodology and present our performance evaluation of our design, including a comparison to SEAL homomorphic encryption. Section 8.5 discusses the security of our optimizations. We discuss the trade-offs of our proposed design in Section 8.6 before concluding in Section 8.7

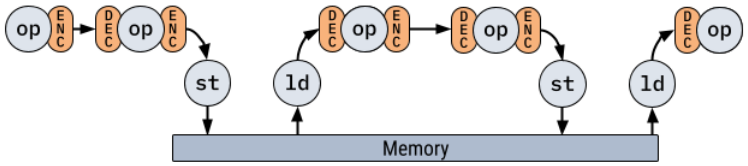
8.2 Sequestered Encryption Optimized Microarchitecture

In Chapter 7, we presented Sequestered Encryption (SE) and its stateless microarchitecture design (Figure 7.2). While this design is uncomplicated, cryptographic operations on the critical path significantly increase the overhead of SE instructions. With QARMA's 12-cycle cipher latency, an add instruction takes 25 cycles to compute in SE [10]. Our evaluation shows that performance overheads are significant, at 12× for QARMA.

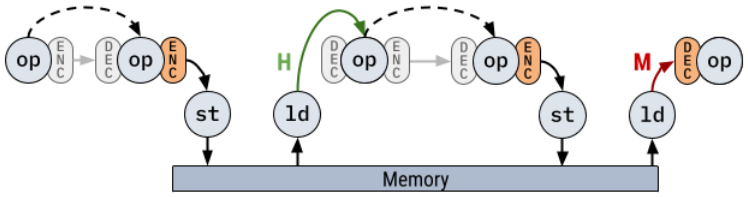
In this section, we present optimizations to the SE design that sufficiently mask these cipher latencies. Our stateful optimized implementation, denoted **SE-Opt**, iterates on the stateless design that serves to reduce performance overheads by masking the encryption and decryption latencies within the SE unit. Other than the SE unit implementation, the remaining SE architecture (*e.g.*, ISA extension) is adopted from Chapter 7. We briefly revisit our security goals to discuss how they guide our architectural optimizations before detailing them in Section 8.2.2.

8.2.1 Security Goals

Our goal for this chapter is identical to that of Chapter 7: to protect of secret user data from disclosure and timing side channels, where secret and non-secret data is characterized by the following definitions:



(a) **Data Flow for Stateless Design.**



(b) **Data Flow for Optimized (Stateful) Design.**

Figure 8.1: **Data Flow for Sequestered Encryption.** In the unoptimized microarchitecture, cryptographic latencies are exposed on every operation. Our optimizations bypass idempotent encrypt-decrypt operations and reduce latencies on the load path.

- ***Secret Data:*** Plaintext values of program variables denoted as Private
- ***Non-Secret Data:*** Ciphertext values of program variables denoted as Private; Plaintext values of program variables not denoted as Private; Constant values and instruction immediates; Instructions and instruction properties (*e.g.*, data dependencies, instruction addresses)

Under these goals, our performance optimizations warrant careful consideration as to not violate secret-data-dependent timing. Our design of SE in Chapter 7 implements the encryption engines and functional units (*e.g.*, ALU) with deterministic timing (*e.g.*, there are no optimized paths for multiply) to thwart timing side channels. However, performance optimizations traditionally introduce timing variations—caches accelerate memory accesses based on spatial and temporal locality, speculation accelerates the execution of predictable branches, *etc.* Our optimizations cannot introduce timing variations based on *secret data*. However, timing variations based on non-secret data (*e.g.*, code reuse or data dependencies) does not violate our security goals. We leverage this in our design.

8.2.2 Optimized (Stateful) SE Unit Microarchitecture

The SE unit microarchitecture, denoted SE, presented in Chapter 7 places encryption latencies on the critical path. This data flow is given by Figure 8.1a. When true data dependencies

occur (*e.g.*, `enc_add x ←x,r`; `enc_add y←y,x`), the result of the first instruction (`x`) is encrypted then immediately decrypted for use in the second instruction. These encrypt-decrypt operations exist on the critical path for neighboring instructions, bloating performance. Further, we expect these dependencies to occur frequently due to locality in the register file.

Our optimizations bypass idempotent cryptographic operations on the critical path, as shown in Figure 8.1b. In SE-Opt, we forwarding previously-computed plaintext values to subsequent dependent instructions within the SE unit. Such forwarding introduces timing variations in the SE design based on instruction characteristics. These differences do not violate our security goals as they are not dependent on secret data. Below, we detail our microarchitectural optimizations.

8.2.2.1 Forwarding Around Decryption

To mask decryption latencies, we extend the SE unit with a small ***decryption cache*** that stores ciphertext-plaintext pairs. Specifically, when an instruction is dispatched to the SE unit, the ciphertext value of a source operand is used to index the decryption cache, which returns the corresponding decrypted plaintext if present. If the decrypted plaintext is not present, the source operand is decrypted in its entirety, as done previously. This design imbues the SE unit with short-term memory of recent decryption operations. Decryption cache latency is dependent on ciphertext values and their reuse, which is non-secret, upholding our security goals related to timing discrepancies.

The decryption cache is updated at two points: *i)* When there is a cache miss and *ii)* When a new value is encrypted by the SE unit. The former optimizes instructions that reuse the same input (`r`) without updating that input (*e.g.*, `enc_add x ←x,r`; `enc_add y←y,r`). The latter optimizes instructions with true data dependencies (*e.g.*, `enc_add x ←x,r`; `enc_add y←y,x`). To accelerate instructions that succeed a load, we dispatch the load to the SE unit to prime the decryption cache so that the ciphertext-plaintext pair is available when the subsequent usage occurs.

We implement the decryption cache as a two-way set-associative cache with eight sets and least-recently-used (LRU) replacement. The cache stores 125-bit ciphertext tags and 8B plaintext data blocks.[1] This small <0.4 kB cache can be accessed in a single cycle and is flushed whenever the SE unit performs a key switch.

[1]To reduce the data block size, we do not store the plaintext salt value because it is immediately discarded after decryption.

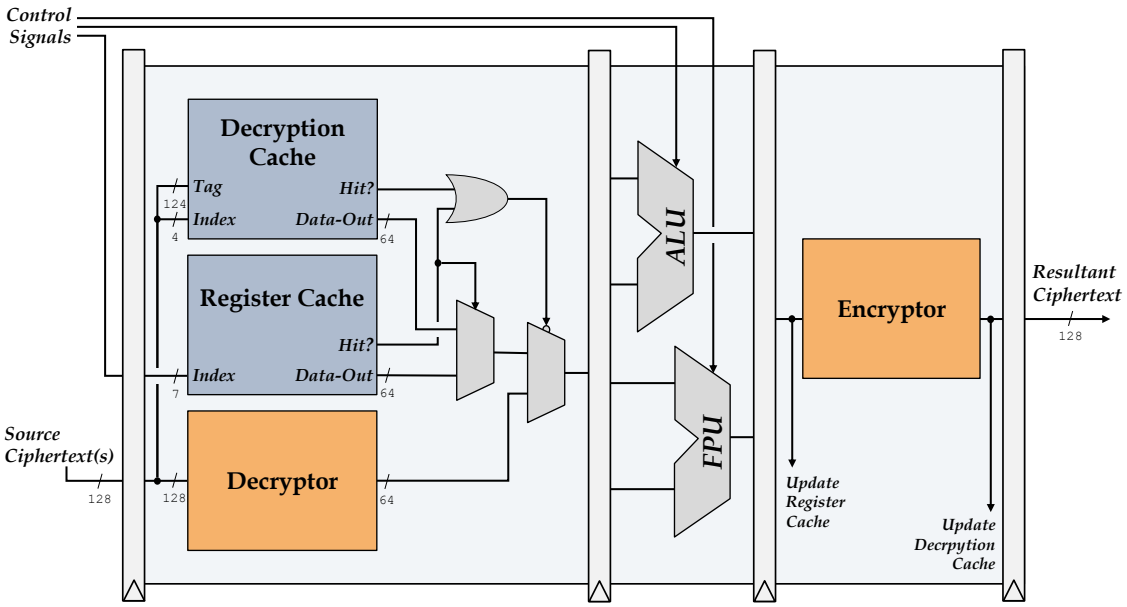


Figure 8.2: **Optimized Sequestered Encryption Microarchitecture.** The decryption cache and register cache forward plaintext values to dependent SE instructions, bypassing idempotent cipher operations on the critical path. Since these caches are indexed with non-secret information (*i.e.*, ciphertext values and register IDs), any timing variation due to these optimizations will not reveal any secret data.

8.2.2.2 Forwarding Around Encryption

To further optimize performance, we add state to internally forward values around the encryptor to subsequent SE instructions. This forwarding occurs *within* the SE unit[2] and can save up to 40 cycles of latency between dependent operations (the latency of AES). We extend the SE unit with an additional decryption cache, termed the ***physical register file (PRF) decryption cache***, or register cache, for short. This new cache is indexed by the physical register identifier[3] of the source operand and stores the corresponding plaintext. This enables forwarding between two instructions that write then read the same physical register while these instructions are both in-flight in the pipelined SE unit. Indexing the cache with the physical register identifier provides two advantages: *i)* It is easy to maintain coherence for the register cache, even during speculation, due to the native design of the PRF, and *ii)* Physical register use is non-secret, upholding our security goals related to timing properties.

The register cache is updated whenever a new plaintext result is computed by the

[2]All values that leave the SE unit pass through the encryptor, per our security goals.

[3]Physical registers are distinct from architectural registers, which are the register names programs and compilers use.

SE computation hardware (*i.e.*, ALU or FPU). For example, after the instruction `enc_add p3←p1,p2` is computed in SE's internal ALU, the register cache entry corresponding to physical register `p3` is updated with the plaintext result. On this cycle, the SE unit also broadcasts that the register `p3` is available so that any dependent instructions can be issued to the SE unit.

We implement the register cache as a direct mapped cache with 128 entries (the size of the PRF) and 8B plaintext data blocks. This 1kB cache can be accessed in one cycle and, as for the decryption cache, is flushed on context switches. The register cache has to be coherent with the PRF, as the PRF may be updated outside of the SE unit. To achieve this, the SE unit snoops the PRF and invalidates cache entries when writes occur outside of the SE-unit (*i.e.*, by the main pipeline).

8.2.2.3 Walkthrough

The optimized miroarchitecture is shown in Figure 8.2. When an encrypted source operand enters the SE unit, the decryption cache and the register cache are accessed in parallel. On a hit in either cache, the plaintext value is read directly from the corresponding cache, bypassing the decryptor. If both caches miss, the ciphertext is passed to the decryptor, as done in SE (§7.3.3). Then, the plaintext result is forwarded to the computation hardware. The caches are updated according to the descriptions above.

Revisiting Figure 8.1, the dashed lines represent forwarding facilitated by the register cache, which optimizes instructions with true data dependencies. The decryption cache primarily serves to bypass decryption operations for values that are stored to memory and re-loaded later. The solid green line represents forwarding facilitated by the decryption cache during a cache hit (H), indicating that the ciphertext has been used recently. The solid red line represents a miss (M), which is likely if there is a long delay between ciphertext accesses.

8.3 Experimental Methodology

We analyze the optimized SE unit microarchitecture, denoted SE-Opt, on a x86-64 out-of-order core in gem5 [6] in system call emulation mode with a 2.5 GHz clock frequency. We extend the core with: *i)* An SE-Opt functional unit in the execute stage (§8.2.2), *ii)* Extensions to the x86-64 ISA that dispatch the SE ISA instructions to the SE-Opt unit (§7.3.1.2), and *iii)* An extended 128-bit register file (§7.3.1.2). The encryption schemes and workloads used for our analysis are the same as Chapter 8 to allow for commensurate comparisons. We restate this evaluation methodology below.

Core Type	2.5GHz, O3CPU (Out-of-Order)
Superscalar	4-wide
Cache Line Size	64B
L1 Cache	32KB with 2-cycle latency
L2 Unified Cache	256KB with 20-cycle latency

Table 8.1: **Experimental Configuration for gem5.** The specifications of our baseline x86-64 out-of-order core in gem5 [6].

Evaluated Data Encryption Schemes. In this work, we evaluate three symmetric encryption schemes: *i)* AES-128 (40 cycles), *ii)* Simon-128/128 (20 cycles), and *iii)* QARMA$_{11}$-128-σ_1 (12 cycles). We use a Mersenne Twister pseudorandom number generator (PRNG) [236] to generate cryptographic salts, re-seeded with a true random source every 200 uses. We used existing open-source encryption libraries within our simulation [237, 238, 239] and parameterized gem5 with the above latencies.

Compiler Support and Benchmarks. We evaluated SE on 21 benchmarks from the VIP-Bench privacy-focused benchmark suite compiled using GNU g++ with -O2 optimizations [15]. VIP-Bench annotates secret variables in its native workloads, enabling us to easily port all annotated VIP-Bench variables to use the encrypted types defined by SE's C++ library (§7.4). Furthermore, VIP-Bench includes both native and data-oblivious variants of each workload. This feature enables us to evaluate SE's overheads compared to a native (unprotected, non-data-oblivious) baseline, as well as analyze the software overheads associated with data-oblivious programming transformations.

8.4 Experimental Results and Evaluation

We assess the performance overheads of SE-Opt against a native baseline. Figure 8.3 shows the performance slowdowns for SE-Opt compared with SE (Chapter 7) using the QARMA cipher. All performance results are enumerated in Table 8.2.

8.4.1 Runtime Performance Overheads of SE-Opt

Our proposed optimizations significantly reduce the performance overheads caused by exposed SE encryption latencies. Average performance slowdowns incurred by SE-Opt are 2.5×, 2.1×, and 2× for AES-128, Simon, and QARMA, respectively. SE-Opt is still latency-sensitive to some degree, especially for applications that frequently stall for stores. But, in

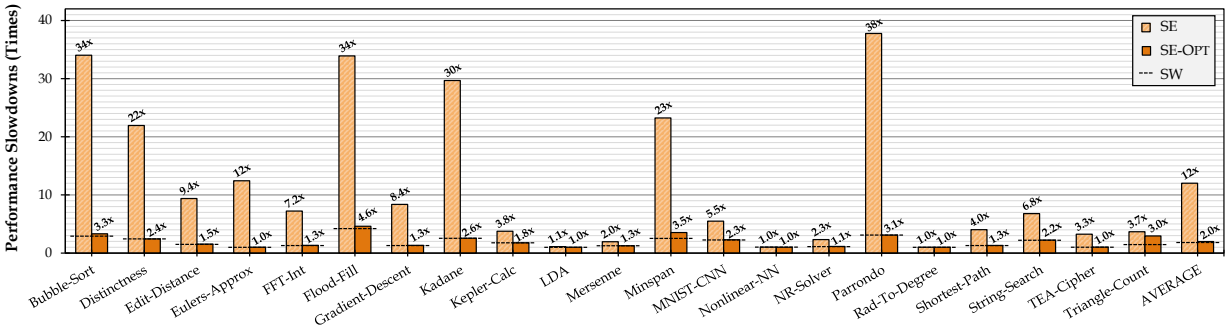


Figure 8.3: **Performance Slowdowns for Sequestered Encryption with QARMA$_{11}$-128-σ_1.** This figure shows the overall performance slowdowns for the stateless SE microarchitecture (SE) and the optimized SE microarchitecture (SE-Opt) when parameterized with 12-cycle QARMA data encryption. The dashed lines represents slowdowns originating from software modifications alone.

general, the cryptographic latencies are largely hidden, making the use of longer-latency ciphers more tolerable. Our optimizations are particularly effective for benchmarks with extensive register-register arithmetic, like *Eulers-Approx*. In these instances, cipher latencies are completely masked, resulting in little differences in performance across encryption schemes.

Our optimizations are so performant that the slowdowns incurred by SE-Opt begin to approach those caused by software modifications alone (*i.e.*, the modified executable running on a x86 core where SE encryption/decryption is disabled). The dashed lines in Figure 8.3 represents software overheads. We see that SE-Opt overheads approach that of software for most benchmarks. More specifically, SE-Opt with QARMA only incurs a 17.5% performance degradation, on average, atop software slowdowns. This demonstrates that the cipher latencies in the SE unit are almost completely masked by our optimizations.

Workloads that exhibit poor register locality and frequent stores still face unavoidable encrypt/decrypt latencies on the critical path, despite our optimizations. Specifically, our optimizations do not entirely mask the cipher latencies for *Minspan* and *Triangle-Count*, which have approximately a 100% and 150% performance degradation from hardware changes, respectively. Other benchmarks that have many short, true data dependencies receive significant benefits from our changes. Many workloads receive at least an order-of-magnitude speedup from SE-Opt compared to the SE design.

Trends and Takeaways. With SE-Opt, overall performance overheads approach those from software alone, informing future directions for further reducing overheads. We hypothesize that the use of 128-bit data types significantly contributes to software overheads (*e.g.*, Booleans are expanded by 16×). Variable data type sizes, reduced salt length, or

		SE			SE-Opt		
Benchmark	Software	AES-128	Simon	QARMA	AES-128	Simon	QARMA
Bubble-Sort	188.9%	10,586.4%	5,382.7%	3,303.1%	390.7%	268.9%	231.4%
Distinctness	142.2%	6,657.0%	3,398.4%	2,094.4%	143.6%	143.4%	143.4%
Edit-Distance	46.5%	2,735.2%	1,377.1%	839.2%	74.3%	59.9%	53.7%
Eulers-Approx	0.0%	3,929.0%	1,939.6%	1,143.9%	0.0%	0.0%	0.0%
FFT-Int	28.7%	2,044.6%	1,024.7%	622.1%	48.8%	34.2%	32.2%
Flood-Fill	324.5%	10,347.0%	5,302.4%	3,291.1%	583.5%	423.3%	360.7%
Gradient-Descent	24.8%	2,609.5%	1,271.8%	736.7%	32.9%	32.9%	32.9%
Kadane	155.5%	9,230.6%	4,685.4%	2,868.2%	155.5%	155.5%	155.5%
Kepler-Calc	73.8%	794.1%	422.8%	275.2%	82.4%	75.8%	76.7%
LDA	0.2%	36.9%	17.0%	9.5%	0.7%	0.4%	0.0%
Mersenne	23.8%	300.2%	152.0%	94.9%	31.2%	28.7%	25.1%
Minspan	152.6%	7,084.7%	3,611.5%	2,224.4%	572.5%	352.3%	253.4%
MNIST-CNN	126.4%	1,623.1%	781.5%	448.2%	159.2%	132.3%	129.2%
Nonlinear-NN	0.8%	8.8%	3.3%	1.5%	0.8%	0.8%	0.8%
NR-Solver	11.0%	428.9%	216.9%	132.4%	29.9%	19.4%	13.7%
Parrondo	204.4%	11,923.1%	6,033.7%	3,677.9%	208.7%	208.7%	208.7%
Rad-To-Degree	0.3%	1.9%	0.7%	0.3%	0.3%	0.3%	0.3%
Shortest-Path	27.6%	967.0%	493.0%	302.0%	30.9%	30.8%	30.0%
String-Search	121.1%	1,668.9%	889.5%	577.5%	128.0%	124.3%	123.9%
TEA-Cipher	2.1%	756.2%	376.5%	225.0%	3.7%	2.3%	2.5%
Triangle-Count	46.6%	1,037.7%	485.8%	265.5%	540.6%	293.8%	195.4%
AVERAGE	81.0%	3,560.5%	1,803.2%	1,101.6%	153.2%	113.7%	98.5%

Table 8.2: **Performance Overheads.** In this table, we show the slowdowns for our SE and SE-Opt microarchitecture designs when parameterized with AES-128 (40 cycles) [10], Simon-128/128 (20 cycles) [13], and QARMA_{11}-128-σ_1 (12 cycles) [10] data encryption. The second column reports slowdowns resulting from only the software modifications required to support SE (*i.e.*, SE encryption/decryption is disabled in the SE unit).

vectorized types (*i.e.*, SIMD) could reduce these overheads. SE could use techniques like ORAM [105] to permit secret-dependent memory accesses, thereby reducing data-oblivious overheads [109]. Compilers can also help programs take better advantage of our proposed architectural optimizations (*e.g.*, prioritized register allocation). These are promising directions for future work. Further, data-oblivious programming modifications may severely degrade performance for some applications. We characterize these overheads and discuss ways to mitigate them in Chapter 9.

8.4.2 Comparison to Homomorphic Encryption

To measure performance benefits compared to HE, we evaluate both frameworks using the workloads presented in Porcupine [14]. These workloads consist of short computation kernels (*e.g.*, dot product) that have limited multiplicative depth, computing only a few operations per data element. For this reason, we do not include these workloads in our main evalua-

tion. We chose Porcupine because these kernels have been optimized specifically for SEAL BFV [244], enabling us to directly run state-of-the-art HE executables with no modifications. We ran the SEAL workloads in a native gem5 instance and compared these results to the SE-Opt microarchitecture parameterized with AES encryption running the same Porcupine computation sequences. Our evaluation, shown in Table 8.3, finds that ***SE provides a speedup over SEAL HE by three orders of magnitude***. Furthermore, we note that the HE workloads evaluated here do not perform bootstrapping. Bootstrapping eliminates noise buildup in computations with significant multiplicative depth and is regarded as very expensive—the most expensive operation in HE. For this reason, bootstrapping is currently not implemented by many HE libraries, including SEAL. If these kernels were to have greater multiplicative depth, bootstrapping requirements would greatly increase HE's overheads, amplifying SE's advantage.

	HE		SE-Opt		SE-Opt
Kernel	Cycle Count	Slowdown over Native	Cycle Count	Slowdown over Native	Speedup over HE
Gx	537,902,255	24,385×	35,722	1.6×	15,058×
Hamming Distance	588,518,480	3,621×	983,636	6.1×	598×
Polynomial Regression	750,253,318	2,095×	1,019,353	2.8×	736×
Roberts Cross	1,006,490,559	34,954×	34,563	1.2×	29,120×
Dot Product	335,496,620	5,364×	763,955	12.2×	439×
Linear Regression	380,780,634	1,354×	803,238	2.9×	474×
AVERAGE		**11,962**×		**4.5**×	**7,738**×
GEOMEAN		**6,006**×		**3.2**×	**1,851**×

Table 8.3: **Comparison to Homomorphic Encryption.** This table presents the processor cycle count and relative performance overheads for the Porcupine kernels [14] when running on SEAL BFV and on SE-Opt parameterized with AES encryption.

8.5 Security Analysis

With modifications to SE's microarchitecture, concerns about data privacy shift toward the timing characteristics of the SE computation. In this section, we discuss how the ciphertexts produced by the SE unit are resistant to information leakage via timing analysis.

8.5.1 Resilience to Timing Analysis

By physically isolating sensitive plaintexts from the main core, we are protected from the many timing side channels innate to modern microarchitectures (*e.g.*, BTB [100], store

buffer [120, 262]). While these side channels still exist in the untrusted core, they only reveal timing dependent on *ciphertexts*. Thus, they cannot be used to violate the confidentiality of sensitive data.

Per our threat model, attackers can also observe timing properties of the SE unit in an attempt to learn secret data. SE is resilient to timing attacks by design, as all instruction semantics exhibit deterministic timing for both microarchitectural designs. Consider the following equation that specifies the timing properties of SE-Opt with the register cache (RC) and decryption cache (DC):

$$\mathrm{L}(\texttt{op}) \;=\; \begin{cases} 1 & \text{if } \texttt{reg id} \in RC, \\ 1 & \text{if } \texttt{ctx} \in DC, \\ \mathrm{L}(\texttt{DEC}) & \text{otherwise.} \end{cases} \;+\; \begin{cases} 1 & \text{if op is ALU,} \\ 3 & \text{if op is FPU} \end{cases} \;+\; \mathrm{L}(\texttt{ENC}) \tag{8.1}$$

where $\mathrm{L}(\cdot)$ denotes the latency of an operation. By construction, these latencies are not plaintext- or key-dependent. The first term of Equation 8.1 depends on whether the register ID or the ciphertext stored in that register is in the register cache or decryption cache. The presence of these values depend on the program execution prior to the current instruction. Since the program is data-oblivious, this execution and therefore the cache contents are independent of secret data. Thus, this timing difference also secret-independent. The second term of Equation 8.1 depends on the opcode of the current instruction. For the same reasons, the execution of this opcode is secret-independent as is this timing difference. While it is trivial to reason that the SE ISA has data-independent timing, a vulnerable or buggy microarchitectural implementation could violate this proof or induce unforeseen timing side channels. Further assurances about side channel resilience can be provided via verification of a digital circuit design.

8.5.2 Potential for Formal Verification

Unlike other hardware security mechanisms, the SE TCB is small and independent of surrounding microarchitectural details, lending itself to formal verification.[4] This minimal design lends itself to formal verification. Our work in [261] performs a complete formal verification of the SE and SE-Opt architectures to show that the design is secure against direct data disclosures and digital side channels for all possible programs. Specifically, we define

[4]As noted by prior work [216], formally reasoning about enclaves like Intel SGX is presumably infeasible as any proof would have to model all processor features that expose registers. Further, such work would be short-lived as any architectural modifications would invalidate this security proof. In contrast, SE's compact design is independent of many pipeline structures, avoiding these deficiencies.

the security requirements of the ISA of SE architecture and use this specification to define the proof obligations for the hardware. We show how these proof obligations can be successfully discharged using commercial formal verification tools for RTL designs. Through this work, we demonstrate that the optimizations introduced in this chapter do not introduce data-dependent side channels.

8.6 Discussion and Limitations

Our optimizations hide long-latency cipher operations, reducing performance overheads to only 2.5× native execution. This speedup provides a significant advantage over our design in Chapter 7, which has up to 36.6× slowdowns across our analyzed workloads. However, this performance gain comes at the cost of an increased TCB, the storage of plaintext state, and added hardware area. Because this design stores plaintext state, it is paramount that the SE unit is fully isolated from the main processor and is only accessible via its defined interfaces. Formal verification can provide these guarantees and, while more complicated due to these optimizations, such efforts are still feasible as they only require validation for a subset of the overall pipeline. Finally, these optimizations rely on the assumption that ciphertexts and code characteristics like data dependencies are non-secret. This assumption aligns with our threat model but may not hold for all systems.

8.7 Chapter Summary

In this work, we build on Sequestered Encryption (SE) to create an encrypted data architecture that is both secure and performant. By leveraging classical architectural optimizations, we safely forward plaintext data values within the SE enclave to mask nearly all exposed cipher latencies in the pipeline. With optimizations, SE achieves <2.5× performance slowdowns over native execution and exhibits a three orders-of-magnitude advantage over software HE libraries. Further, our observed slowdowns begin to approach those attributed to software overheads. This finding motivates future work to better understand programming overheads and tackle the remaining slowdowns. With the mechanisms presented in this chapter, architectural approaches can emerge as dynamic, expressive, and performant data privacy solutions that possess zero trust in software.

CHAPTER 9

Understanding the Program Implications of Encrypted Data Architectures

In our analysis of Sequestered Encryption in Chapter 8, we found that a majority of performance overheads originated from software modifications. This was especially true when using lightweight ciphers like QARMA, where less than 20% of overheads originated from the encryption latency and other hardware modifications. Motivated by this fact, this chapter conducts the first performance characterization of data-oblivious programs. Data-oblivious programs are relevant beyond SE and have gained popularity due to their application in security. However, approaches requiring obliviousness are often dismissed because of anticipated performance loss. In this chapter, we study the overheads of data-oblivious transformations applied to twenty workloads from the VIP-Bench benchmark suite, and we find that, overall, performance overheads vary widely, with a geomean slowdown of 7.4×. This variance can be attributed to whether or not the data-oblivious transformations affect the workload's asymptotic complexity. Performance overheads are much lower for the fourteen workloads whose complexity is unaffected, at 1.9× geomean. Leveraging lessons from analyzing these overheads, we study four notably slow data-oblivious workloads and show how algorithmic changes can significantly improve performance—achieving an average 86.4× speedup over the mechanically produced baseline programs. While data-oblivious program execution often incurs overheads, the contributions of this chapter show that these overheads can be overcome by compiler and algorithmic optimizations, bringing us closer to achieving efficient and widely-used data-oblivious programs for encrypted architectures.

The research presented in this chapter appears in:

[263] Lauren Biernacki, Biniyam Mengist Tiruye, Meron Zerihun Demissie, Fitsum Assamnew Andargie, Brandon Reagan, Todd Austin. ***"Exploring the Efficiency of Data-Oblivious Programs,"*** IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2023. DOI: 10.1109/ISPASS57527.2023.00027.

9.1 Introduction

In the final work of this dissertation, we explore the program implications of encrypted data architectures. Specifically, techniques that encrypt data, such as SE, require programs to be data-oblivious—program control flow or memory accesses cannot depend on the encrypted types. The overheads associated with data-obliviousness are thought to be prohibitive. Thus, techniques that require it are often dismissed. Further, our analysis of SE finds that a majority of our runtime overheads arise from software changes.[1] These reasons motivate the study of data-oblivious workloads, as it can promote the adoption of SE and further address its overheads.

Data-oblivious programming is contrary to high-performance coding practices that often intermingle program execution and input values via data-dependent program heuristics and data structures. Fundamental programming constructs like if-statements, loop conditions, and array indexing are often influenced by program inputs. Coupling a program's execution with its input data values in this way is often intuitive and higher performing. However, many issues we face in computing today are, in part, artifacts of this method of developing code. In fact, data-oblivious programming originated from the security community far before SE. Side-channel attacks (§2.5.3) have been used to break cryptosystems [39, 97, 98, 99] and, more recently, exfiltrate data from commercial enclaves like Intel SGX [118, 221]. These attacks observe a program's data-dependent execution behavior to exfiltrate the secrets the program is processing. A ***data-oblivious program*** decouples its data inputs from program execution, exhibiting control flow and memory access patterns that are independent of its input data. This technique has gained attention in the security community to guard against side-channel attacks [109, 227, 228, 229, 264, 265], as observation of either the CPU or memory system renders zero information about the data inside [266].[2]

To date, the major drawback of data-oblivious programming is performance loss. Eliminating dynamic control flow requires the predication of conditional execution paths. Further, implementing input-data-dependent memory lookups requires entire data structures to be accessed. Finally, all loops must be executed using their maximum trip count, so that loop execution does not reveal program values. Each of these conversions introduce extra, superfluous computation to mask input-dependent behavior. It is widely believed that these

[1] When using AES-128 encryption, SE achieves performance that is within 154% (2.5×) of native execution, with 73% of this overhead due to cryptographic latencies and the remaining 81% due to software overheads.

[2] Data-oblivious programming is sometimes called "constant-time" programming in security disciplines. We avoid this term because a constant-time program can still exhibit data-dependent control flow or memory accesses.

code transformations introduce overwhelming runtime overheads, limiting the adoption of data-oblivious programs.

This chapter presents the first detailed characterization of data-oblivious programs using a large collection of benchmarks from VIP-Bench [15]. We apply standard data-oblivious transformations (*i.e.*, if-conversion, loop-conversion, and access-conversion) to these workloads and measure the relative performance impact experienced by each. Across the entire suite, we find that overheads are highly variable, with some benchmarks experiencing minor slowdowns and others incurring overheads of three orders of magnitude. Despite this, we find the per-instruction performance of the benchmarks improves, with branches being more predictable and cache hit rates improving overall. Data-oblivious programming imbues algorithms with a level of predictability and determinism that can be exploited by existing architectural optimizations.

To address the workloads experiencing high slowdowns, we examine the utility of redesigning these benchmarks' core algorithms under a data-oblivious model of computation. We show how changing the algorithm used to solve a particular problem can mitigate data-oblivious overheads. In particular, our insight is that oftentimes, the algorithm optimized for a non-oblivious setting is not necessarily optimal in a data-oblivious setting. For example, the string search algorithm *Boyer-Moore-Search*, which uses heuristics to achieve a best-case complexity of $O(n/m)$, incurs over a 1200× slowdown when made oblivious. This is in sharp contrast to a naïve string search implementation, which incurs a 1.5× slowdown over the same baseline. Across our case studies, our algorithmic optimizations provide a geomean speedup of 86.4× compared to mechanically converted data-oblivious programs. This result moves us closer to achieving efficient and widely-used data-oblivious programs for encrypted architectures and beyond.

9.1.1 Chapter Contributions

In summary, this chapter makes the following contributions:

- We present the first characterization of data-oblivious programs via a detailed performance analysis of 20 benchmarks from VIP-Bench [15].
- We conclude that the overheads of data-obliviousness are highly variable, with a geomean slowdown of 7.4×, and highlight the competing factors for slowdown and speedup. These results motivate compiler and hardware research to address the remaining overhead.
- We show how benchmarks with extreme overheads may be the result of poor algo-

rithm selection and how algorithm redesign can help reduce data-oblivious program overheads. We revisit four VIP-Bench workloads to recover an 86.4× performance improvement, on average, atop the mechanically converted workloads.

In Section 9.2, we detail data-oblivious program transformations, which we measure using the methodology detailed in Section 9.3. Sections 9.4-9.6 examine the performance implications of data-oblivious execution. Finally, we detail related work and conclude in Sections 9.7 and 9.8.

9.2 Data-Oblivious Transformations

A data-oblivious program is a program whose execution is independent of input data values. We consider a program to be data-oblivious if its control flow and memory access patterns are independent of its inputs. Some programs are naturally data-oblivious, such as those rooted in dense linear algebra or engineered for regularity (*e.g.*, security code). Many more programs are not data-oblivious, as it is most natural for programmers to think in data-dependent terms.

In this section, we review the three primary techniques to convert a program to be data-oblivious: ***If-Conversion***, ***Loop-Conversion***, and ***Access-Conversion***. Through conducting this study, we found that some programs were still not oblivious after applying these transformations due to overlooked language-specific behavior. Specifically, conditional short-circuiting operations were problematic as they induce branches in the resultant binary. We also address these operators below. To gain confidence that our converted workloads were data-oblivious, we ran them with VIP-Bench's Functional Library [15], which provides a software-based implementation of encrypted computation using x86 AES-NI instructions [267]. This validated that input data values did not influence memory addressing or branch conditions and were only used as source operands for the operators listed in Table 9.1. We detail these transformations in the following sections.

9.2.1 If-Conversion

If-conversion serves to address conditionally-executed instructions to ensure that a workload's execution is independent of its input data. Specifically, if-statements with input-dependent conditions violate obliviousness as they introduce data-dependent control flow in the resultant binary. To make these statements data-oblivious, they must be transformed through if-conversion, which removes the control dependencies by predicating the instructions within the statement's basic block. We use the x86-64 CMOVcc primitive to predicate all writes to

live variables within the statement's basic block. If the if-statement contains an else block, any instructions within the else block are similarly predicated with the inverse of the condition. This follows for if-statements containing multiple else-if conditions. The following code demonstrates how a program can conditionally increment `x` or `y` using this technique.

```
1 x = CMOV( cond, x+1, x);
2 y = CMOV(!cond, y+1, y);
```

While these `CMOV` operations appear functionally similar to ternary operators (`?:`), their semantics are different. Ternary operators are short circuited, per the C++ standard, and are compiled to conditional branches. Thus, ternary operators are not data-oblivious. If the x86-64 CMOVcc primitive is not available, data-oblivious conditional logic can also be implemented using arithmetic primitives:

```
1 x = ((cond)*(x+1))  + ((!cond)*(x));
2 y = ((!cond)*(y+1)) + ((cond)*(y));
```

We discuss the specific syntax of `CMOV` in Section 9.3.

9.2.2 Loop-Conversion

Loop statements with input-dependent iterations similarly violate obliviousness as they introduce data-dependent branches. Specifically, loops with a variable number of iterations or early-exit conditions must be transformed through loop-conversion to take a fixed number of iterations. Any condition that previously terminated the loop is transformed into a Boolean value that predicates writes to any variables live outside of the loop (similar to if-conversion). For example, consider the following loop from *Primality-Test*, which exits when `x` reaches one of the specified base cases.

```
1 for(int r=1; r<=s; r++){
2   x = (x * x) % n;
3   if (x == 1)
4       return COMPOSITE;
5   if (x == nm1)
6       break;
7 }
```

Loop-conversion predicates all writes to `x`, ensuring that `x` is not changed after a base cases is reached, and introduces a variable to capture the return statement on Line 3.

```
bool done = false;
int  ret  = -1;
for(int r=1; r<=s; r++){
  x    = CMOV(!done, (x * x) % n, x);
  ret  = CMOV(!done & (x==1), COMPOSITE, ret);
  done = done | (x==1) | (x == nm1);
}
...
return ret;
```

Recursion similarly violates obliviousness since program control flow, in the form of function calls and stack frames, is dependent on input data. During loop-conversion, we unroll recursive code into a while loop. This iterative version is then transformed to be data-oblivious as described above.

Generally, loop conversion negatively affects the time complexity of a program. In *Primality-Test*, this causes the workload to always perform the worst-case number of iterations, making the run time equal to the native variant's worst-case complexity. In workloads that involve recursion, the impacts on time complexity are often more severe.

9.2.3 Access-Conversion

Input-dependent memory access patterns violate obliviousness. These memory patterns often manifest as array accesses where the index depends on input data (*i.e.*, reading from or writing to `arr[idx]`). Access-conversion replaces a single data-dependent array access with reads and writes to the entire data structure while only performing the desired update for the index specified by the non-oblivious operation. For example, the following code demonstrates how to read `arr[idx]` into the local variable `ret`.

```
for(int i=0;  i<n;  i++;)
    ret = CMOV(i==idx, arr[i], ret);
```

This transformation is, effectively, a trivial construction of oblivious RAM (ORAM) [266]. More advanced ORAM techniques perform fewer memory accesses while being statistically secure (*e.g.*, Path ORAM [105]). Analysis of these schemes is out of scope for this work, but we discuss them in Section 9.7.

Like loop-conversion, access-conversion can impact the time complexity of the benchmark by an order of magnitude. These impacts are further exacerbated by the fact that input-dependent array updates are often critical to an algorithm and located within its main loop, exacerbating overheads.

9.2.4 Operators

Data-dependent control flow can be implicit in the semantics of some operators in high-level programming languages. C++ implements logical AND (`&&`) and OR (`||`) operators using short-circuit evaluation—evaluating the first argument and eliding execution of the second argument if its computation is redundant. Effectively, short-circuit operators are control structures, not simple arithmetic operators, and result in the introduction of conditional branches in the workload's executable. Thus, any operator that uses short-circuit evaluation is *not* data-oblivious. In C++, these non-oblivious operators are `?:`, `&&`, and `||`. Our oblivious workloads exclusively use the operators listed in Table 9.1, alongside conditional moves, to make decisions on the input data set. We include the conversion of short-circuited operators under if-conversion in our evaluations, as these transformations effectively remove input-dependent conditional branches from the program binary.

Class	C++ Operators
Arithmetic	`+, -, *, /, %, ++, --`
Assignment	`=, +=, -=, *=, /=, %=`
Relational	`==, !=, <, <=, >, >=`
Bitwise and Logical	`&, |, ^, «, », !`

Table 9.1: **Oblivious Operators in C++.** This table lists the C++ operators included in our oblivious workloads. Short-circuited logical (`&&`, `||`) and ternary (`?:`) operators are compiled via control structures and therefore are not data-oblivious.

9.2.5 Frequency of Data-Oblivious Transformations

In this work, we apply these data-oblivious transformations to the set of VIP-Bench benchmarks [15] to characterize their performance impacts. We do not evaluate one-third of the VIP-Bench workloads because they are *already* oblivious. Further, of our evaluated workloads, only half require loop-conversion and one-fourth require access conversion. This demonstrates that many valuable algorithms are already oblivious, with others requiring only light modifications to achieve obliviousness (typically in the form of if-conversion). We find that many dynamic programming algorithms are nearly data-oblivious as their looping structure is often iterative with a fixed memory access pattern. For example, the Wagner–Fischer algorithm used in *Edit-Distance* updates values in a matrix iteratively depending on the neighboring cells and only requires a small if-transformation to achieve obliviousness. Nevertheless, these generalizations are subject to exceptions. *Knapsack*, which also uses dynamic programming, is burdened by an input-dependent access pattern (*i.e.*, indexes a matrix with an intermediate weight), and thus incurs high overheads due to access-conversion.

9.3 Evaluation Methodology

Below, we detail our approach to assessing the overheads of the data-oblivious transformations detailed in Section 9.2.

9.3.1 Compiler Support and Benchmarks

We evaluate 20 workloads from the VIP-Bench benchmark suite [15], listed in Table 9.2. VIP-Bench provides data-oblivious variants of each workload, enabling us to identify the program transformations required to convert native workloads. We use the following terms in reference to each workload:

Native—The unmodified version of the algorithm.

Oblivious—The data-oblivious version of the algorithm, *i.e.*, the native workload with if-, loop-, and access-conversions applied (by hand) when necessitated.

Ten workloads in the VIP-Bench suite are innately data-oblivious (*i.e.*, their native and data-oblivious variants are equivalent). Thus, we do not include these workloads in our analysis as they do not require any of the program transformations we wish to characterize.

We organize the remaining workloads into two groups, depending on whether or not the data-oblivious transformations affect the workload's asymptotic complexity. In some cases, this effect is severe, with the average time complexity of *Flood-Fill* increasing by a factor of n^2.

Characterizing Individual Data-Oblivious Transformations. To quantify the overheads associated with if-, loop-, and access-conversion, we created variants of each workload that successively apply these transformations. Specifically, we produced four variants of each benchmark: *i)* The native version, *ii)* The native version with loop-conversion, *iii)* The native version with loop-conversion and if-conversion, and *iv)* The native version with loop-conversion, if-conversion, and access-conversion (*i.e.*, the oblivious version). Because loop-conversion modifies conditional logic (*e.g.*, by replacing an early return with a conditional update to a return variable), we apply it first to fully isolate the costs related to if-conversion. We apply access-conversion last as it introduces significant changes to the workload and incurs large overheads. Short-circuited transformations for Boolean operators are included in if-conversion.

Given these four variants, we can deduce the slowdowns attributed to each transformation. For example, slowdown due to if-conversion is computed by comparing the cycle counts for variants *(ii)* and *(iii)*.

	Conversions			Complexity		
Benchmark	If	Loop	Acc.	Native	Oblivious	Input Size
Bubble-Sort	●	●	○	$O(n^2)$	$O(n^2)$	$n = 4096$
Edit-Distance	●	○	○	$O(n^2)$	$O(n^2)$	$n = 160$
FFT-Int	●	●	○	$O(1)$	$O(1)$	-
GCD-List	●	●	○	$O(n)$	$O(n)$	$n = 10,000$
Image-Filter	●	○	○	$O(nm)$	$O(nm)$	$n = 400, m = 600$
Kadane	●	○	○	$O(n)$	$O(n)$	$n = 360$
Kepler-Calc	●	●	○	$O(1)$	$O(1)$	-
LDA	●	○	○	$O(n)$	$O(n)$	$n = 800$
MNIST-CNN	●	○	○	$O(n)$	$O(n)$	$n = 25$
Nonlinear-NN	●	○	○	$O(1)$	$O(1)$	-
NR-Solver	●	●	○	$O(1)$	$O(1)$	-
Parrondo-Game	●	●	●	$O(1)$	$O(1)$	-
Set-Intersection	●	○	○	$O(nm)$	$O(nm)$	$n = 5000, m = 4000$
Shortest-Path	●	○	○	$O(n^3)$	$O(n^3)$	$n = 100$
Boyer-Moore-Search	●	●	●	$O(nm)$	$O(n^2m)$	$n = 512, m = 8$
Distinctness	●	●	●	$O(nlogn)$	$O(n^3)$	$n = 128$
Flood-Fill	●	●	○	$O(n^2)$	$O(n^4)$	$n = 32$
Knapsack	●	○	●	$O(nw)$	$O(nw^2)$	$n = 50, w = 250$
Minspan	●	○	●	$O(n^2)$	$O(n^3)$	$n = 100$
Primality-Test	●	●	○	$O(log^3 n)$*	$O(m^3)$	$m =$ `sizeof`(n)

Table 9.2: **Workloads.** Above, we list our evaluated workloads from VIP-Bench [15]. The ● symbol indicates which data-oblivious transformations are applied to each workload. We report average time complexity for the native and oblivious algorithm variants. The workloads are grouped depending on whether the data-oblivious transformations impact the asymptotic complexity. *The complexity of native *Primality-Test* depends on the value of n itself.

Compiling Workload Executables. All workloads were compiled using GNU g++ with -O3 optimizations. Any oblivious workloads requiring math support were linked with a transformed version of the `mathlib` library. Conditional moves (`CMOV`) were implemented by emitting x86-64 CMOVcc instructions via inline `asm` statements, as shown in Figure 9.1.

Prior work has noted that CMOVcc instructions are not guaranteed to be data-oblivious by the x86 ISA, as they could be implemented in microcode using a branch instruction [109]. While this is not the case in this work, such observations warrant ISA specifications to more explicitly state data-oblivious guarantees, or the lack thereof.

9.3.2 Experimental Setup

We evaluated our workloads on gem5 [6] with an out-of-order core configured to match the performance of the Intel Skylake microarchitecture [268]. This configuration is summarized in Table 9.3. Performing our analysis in a simulated environment enabled us to collect

```
template <class T>
T CMOV(bool cond, T v_true, T v_false){
    T result;
    __asm__ __volatile__(
        "test   %1, %1;"  //Compute bitwise AND
        "cmovz  %3, %0;"  //Move if zero
        "cmovnz %2, %0;"  //Move if not zero
        : "=r" (result)
        : "r" (cond), "r" (v_true), "r" (v_false)
        : "cc" );
    return result;
}
```

Figure 9.1: **Implementation of CMOV.** To implement conditional moves using the x86 CMOVcc primitive, we emit inline `asm` statements that set the destination register `result` with either `v_true` or `v_false`, depending on the evaluation of the Boolean condition.

detailed microarchitectural metrics and eliminate system noise. The processor configuration we employ [268] provides confidence in our results, as it has been validated to match the performance of real hardware.

Core Type	3.5GHz, O3CPU (Out-of-Order)
Superscalar	8-wide
L1 Cache	32KB, 16-way set assoc. with 1-cycle latency
L2 Cache	1MB, 16-way set assoc. with 14-cycle latency
L3 Cache	2MB, 16-way set assoc. with 44-cycle latency
Memory	32GB DDR4

Table 9.3: **Tuned Skylake Processor Configuration for gem5.**

9.3.3 Metrics

We report metrics for each workload's primary computation kernel to measure the impact of data-oblivious transformations on the core algorithm. These measurements do not capture data initialization, result validation, or printing, which can all amortize the costs associated with data-oblivious transformations. Below, we define the metrics used in our evaluation.

Performance Overhead (Cycle Count): Performance overhead is computed by dividing an oblivious workload's processor cycle count by the native processor cycle count. A performance overhead of less than one indicates that the oblivious workload's runtime is actually *faster* than its native counterpart.

Instructions per Cycle (IPC): Instructions per Cycle (IPC) denotes the average number of instructions computed in one processor cycle. In our eight-way superscalar processor, up to eight instructions can be executed each cycle by exploiting instruction-level parallelism. An IPC of less than one indicates that the processor is stalling or squashing instructions due to data, structural, or control hazards.

Dynamic Instruction Count: Dynamic instruction count is the number of instructions committed by the processor for the specific workload. While instruction counts increase for data-oblivious workloads, impacts on branch count vary, as if-conversion removes branches, whereas loop-conversion and access-conversion introduce them.

ALU Utilization: ALU utilization is defined as the fraction of cycles in which each arithmetic logic unit (ALU) accepts a new input. For example, 50% ALU utilization indicates that all of the pipelined ALU units are occupied half of the time, on average.

9.4 Characterization of Mechanized Data-Oblivious Transformations

We characterize the performance overheads and instruction characteristics of oblivious workloads under the mechanised conversion presented in Section 9.2. Table 9.2 lists these workloads, the data-oblivious transformations applied, and any resulting changes in asymptotic complexity. Six workloads experience an increased asymptotic complexity due to data-oblivious conversions (*Boyer-Moore-Search*, *Distinctness*, *Flood-Fill*, *Knapsack*, *Minspan*, and *Primality-Test*). We find that these six workloads exhibit high overheads, at 179× geomean, compared to the remaining benchmarks, at 1.9×. We separate these six workloads in the presentation of our figures and tables. In Section 9.5, we look at the ways in which redesigning applications' core algorithms can avoid these overheads.

All workloads incur an increased ***dynamic instruction count*** from the data-oblivious transformations (10.6× geomean). However, data-oblivious transformations also work to convert control dependencies into data dependencies, which architectures are designed to resolve through out-of-order execution and forwarding. Thus, we observe that many benchmarks exhibit an ***improved IPC***, helping to offset instruction overheads.

Figure 9.2 presents the ***performance overheads*** of the oblivious VIP-Bench workloads. This figure is presented on a logarithmic axis due to the large differences in the measured slowdowns for some workloads. Across all twenty benchmarks, data-oblivious transformations incur a 7.4× performance slowdown. The normalized overheads are presented in Fig-

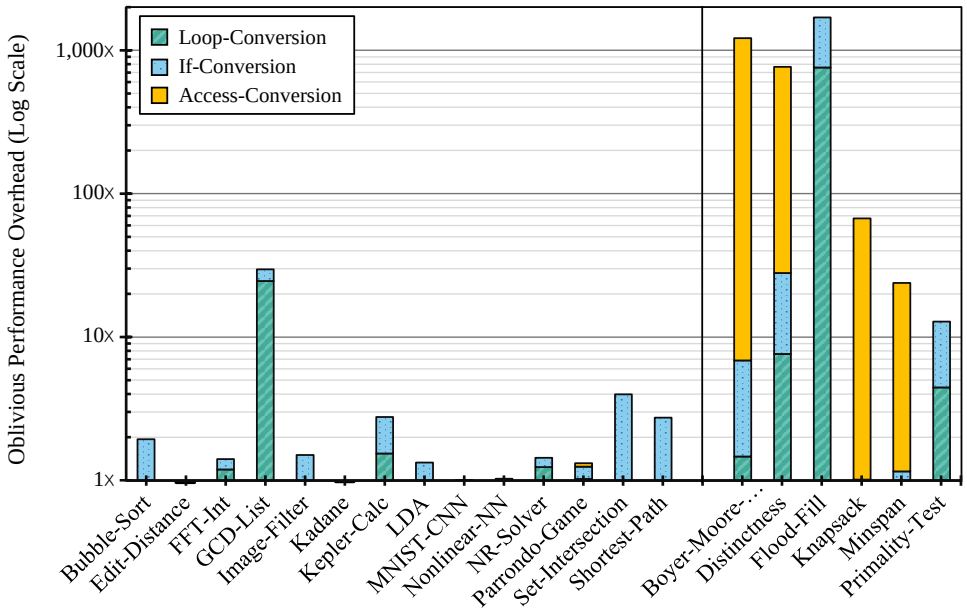


Figure 9.2: **Performance Overhead, Relative to Native (Log Scale).** This figure shows the performance overhead for each oblivious workload, relative to a native baseline.

ure 9.3, illustrating the relative breakdown of the programs' runtime as attributed to native computation, loop-conversion, if-conversion, and access-conversion. Below, we characterize sources of overheads for each of these conversions.

9.4.1 Performance Overhead of If-Conversion

The performance overhead associated with if-conversion varies per workload, ranging from a 4.7× slowdown in the case of *Boyer-Moore-Search* to a slight speedup in the case of *Edit-Distance* and *Kadane*. The geomean slowdown due to if-conversion across all workloads is about 1.6×.

9.4.1.1 Sources of Overhead

If-conversion does not impact the asymptotic complexity of a workload. However, it can significantly degrade performance when the if-statement contains several clauses. In these instances, if-conversion greatly increases the number of instructions and memory accesses executed by the program. Our most egregious example is the math function `exp()`, which is provided by our C++ `mathlib` library. In the native variant, `exp()` executes a large if-statement containing fourteen clauses which check for common input cases before computing the intended exponential function. While these predictions serve to optimize the average

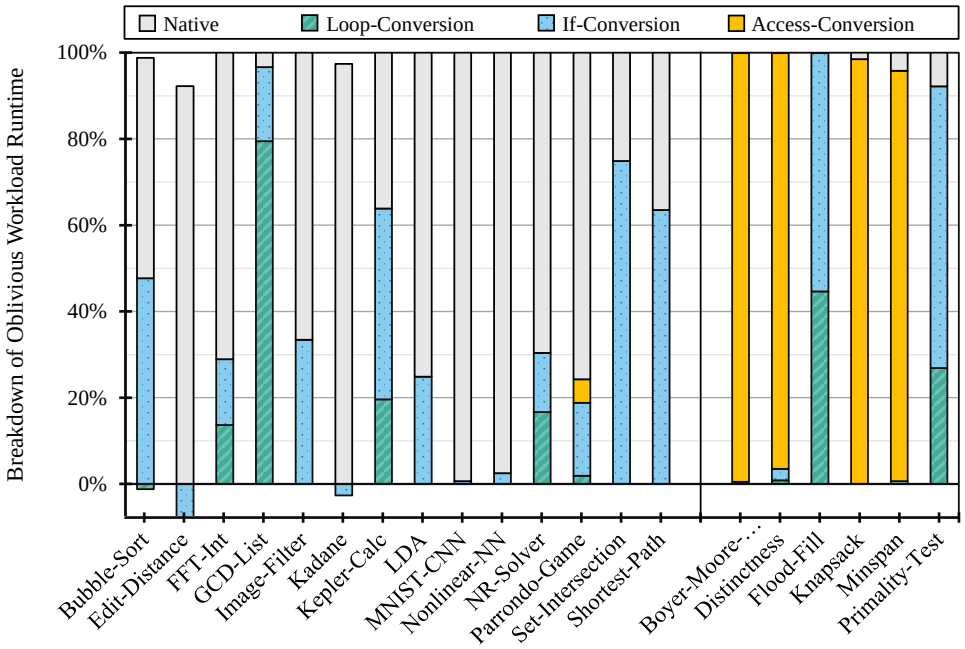


Figure 9.3: **Breakdown of Oblivious Runtime.** This figure shows the percentage of an oblivious workload's runtime attributed to different transformations. For example, 67% of the oblivious *Image-Filter* runtime is attributed to computation in the native algorithm, whereas 33% is attributed to if-conversion. *Edit-Distance* and *Kadane* experience speedups resulting from if-conversion, as noted in Section 9.4.1. *Bubble-Sort* also experiences a speedup from loop-conversion, which removes an early-exit condition, but this speedup is masked by the overheads of other data-oblivious transformations.

performance, if-conversion of this function amasses fourteen predicates.[3] *Primality-Test* similarly includes multi-clause if-statements to detect base cases, an excerpt of which is given in Section 9.2.2. As a result, its if-conversion overhead is moderate at 2.9×. This trend follows for *Boyer-Moore-Search* and *Distinctness*, which similarly have multiple predicates.

If-conversion also degrades performance when a large proportion of a workload's logic is conditional. Figure 9.3 shows that processor cycles induced by if-conversion account for at least half of the overall workload runtime for *Bubble-Sort*, *Shortest-Path*, and *Set-Intersection*. In the first two cases, if-conversion introduces additional stores per loop iteration, increasing the overhead relative to the other (trivial) loop logic. *Set-Intersection*'s main logic is similarly simple, consisting of only a single if-statement within a nested for-loop. Because of this high proportion of conditional logic, *Set-Intersection*'s if-conversion doubles the number of instructions executed and causes a 4× performance degradation.

If-conversion can also benefit workloads. If-conversion has long been used in compiler

[3]The only VIP-Bench workload that uses `exp()` is *MNIST-CNN*, which calls it sparingly when computing sigmoid.

optimizations to improve code performance by selectively replacing control dependencies with data dependencies [269]. In the case of *Edit-Distance* and *Kadane*, if-conversion produces an oblivious executable that is more performant than its native counterpart, with speedups of 8.4% and 2.7%, respectively. However, compilers leverage heuristics to determine which branches will benefit from if-conversion. Since we transform all input-dependent conditionals, not all of our workloads experience a speedup.

9.4.1.2 Takeaways

Of all oblivious transformations, if-conversion imposes the lowest overheads. We find that if-conversion incurs more significant overheads when the benchmark *i)* Exhibits complex if-statements with multiple predicates, or *ii)* Has a high proportion of conditional logic compared to other native computation. Case *(i)* occurred sparingly in VIP-Bench, whereas case *(ii)* occurred for the most simplistic workloads, such as *Set-Intersection*. Only five of the twenty workloads requiring if-conversion incurred a slowdown of over 2×.

9.4.2 Performance Overhead of Loop-Conversion

Loop-conversions fall into two categories: *i)* The loop is changed to a constant number of iterations, either through modifying the loop condition or removing an early exit, or *ii)* The control logic is changed more significantly, affecting the asymptotic complexity. Our observed runtime overheads reflect the severity of these two cases, with a geomean 2.0× overhead in the former case compared to a slowdown of over 14× in the latter. Across all workloads, loop-conversions incur an geomean 4.3× slowdown.

9.4.2.1 Sources of Overhead

Two workloads, *Distinctness* and *Flood-Fill*, require significant loop-conversion modifications that erode their asymptotic complexity. The severity of these changes results from their hostile control logic. *Distinctness* checks for uniqueness by inserting numbers into a binary search tree (BST). Traversal of a BST, which can be done in $O(logn)$ time, is not data-oblivious and must be replaced by a constant $O(n)$ loop that assesses every node. This change incurs a 7.6× slowdown. The native *Flood-Fill* algorithm is implemented using recursion. Mechanically converting the functional logic of *Flood-Fill* for an oblivious setting requires the insertion of two outer loops, increasing the workload's time complexity from $O(n^2)$ to $O(n^4)$. The workloads performance overhead reflects this significant degradation in time complexity, incurring a 759× slowdown from loop-conversion alone. *Boyer-Moore-Search* and *Primality-Test* also incur an increased complexity resulting from loop-conversion

and have a 1.5× and 4.5× slowdown, respectively.

For the other six workloads that experience loop-conversion, changes involved removal of an early-exit (*e.g.*, *Bubble-Sort*) or setting a loop to a fixed number of iterations (*e.g.*, *Kepler-Calc*). These transformations either cause the workload to have the worst-case time complexity or increase the workload's complexity by a constant. In this group, two workloads are of particular interest. Although data-oblivious conversion does not increase the complexity of *GCD-List*, the workload does have significant overheads resulting from loop transformations. The application is modified to include a constant sized loop (32) that wraps the main computation. Thus, we see the overall slowdown of *GCD-List* begin to approach this constant, at ∼30×. *Bubble-Sort* experiences a slight speedup of ∼2% because loop transformations remove an early-exit. While this causes the loop to execute for more iterations, it removes work from within the loop condition, causing this change in the runtime. Slowdowns from if-conversion mask this speedup, but it is evident in Figure 9.3.

9.4.2.2 Takeaways

In some instances (*e.g.*, data structure traversal), program control-flow must be significantly modified to adhere to an oblivious setting, causing high overheads. Rethinking these algorithms to use more intentional control flow, rather than mechanically converting native code, can work towards reducing these overheads. In Section 9.5, we explore how *Flood-Fill* and *Distinctness* can be redesigned to avoid much of the loop-conversion overheads we describe above.

9.4.3 Performance Overheads of Access-Conversion

Access-conversion is only required by five VIP-Bench workloads. When needed, it significantly degrades performance, with an average 23.4× slowdown. This overhead stems from the additional memory accesses required by these program conversions, which replace a single array lookup with accesses to the entire data structure.

9.4.3.1 Sources of Overhead

Access conversion increases a workload's complexity by a factor equal to the length of the accessed structure. In *Parrondo-Game*, access-conversion is used to update an array with a static length of three, and therefore has less impact on the program's runtime. For the other four workloads, access conversion impacts asymptotic complexity by a factor of the input size (n) and increases memory accesses by an order of magnitude.

As illustrated by Figure 9.3, access conversion accounts for nearly all the runtime of oblivious *Boyer-Moore-Search*, *Distinctness*, *Knapsack*, and *Minspan*. Performance overheads resulting from these conversions are high, at 176.9×, 27.4×, 66.3×, and 20.6×, respectively. In some instances, the overhead of access conversion is compounded by other data-oblivious modifications. For example, *Flood-Fill*'s time complexity increases from $O(n^2)$ to $O(n^4)$ over all transformations, incurring a whopping 1697× slowdown over native.

9.4.3.2 Takeaways

Access-conversion is inefficient since it requires a trivial ORAM scan (Section 9.2.3) over the entire corresponding data structure. However, these costly scans can sometimes be avoided. While input-dependent memory lookups are central to many algorithms, we have found instances where workloads can be re-written to subvert this logic. Such is the case with *Boyer-Moore-Search*. In Section 9.5, we discuss changes to this algorithm that avoid ORAM scans to effectively reduce the oblivious overheads from 1214× to under 2×. If input-dependent memory accesses are inevitable, more advanced ORAM mechanisms have the potential to reduce slowdowns. Evaluation of these advanced ORAM schemes (discussed in Section 9.7) are out-of-scope for the goals of this work.

9.4.4 Trends in Instruction Characteristics

The absolute costs of data-oblivious programming can be expensive for some workloads, whereas others are more amenable to oblivious transformations. Performance overheads generally degrade as a result of if-, loop-, and access-conversion because the workloads' dynamic instruction count and memory accesses increase. Despite these costs, we find that the behavior of individual instructions improve. These trends are illustrated in Figure 9.4, where the last four rows show trends in instruction characteristics.

As data-dependent branches are removed, program branching patterns become more predictable. As a result, the branch misprediction rate improves for a majority of our workloads. Memory accesses also become more predictable and can exhibit better spatial locality (*e.g.*, *Bubble-Sort*), which is evidenced by improved miss rate in the L1 Data Cache. ALU utilization similarly improves for a majority of our workloads as a result of the removal of branches and pipeline stalls.

Further, we see an improvement in IPC due to data-oblivious transformations for a majority of our workloads. This improved IPC serves to partially counter the absolute costs associated with data-oblivious modifications. For example, the dynamic instruction count for *Flood-Fill* increases by almost 4680× due to data-oblivious conversions. But, the work-

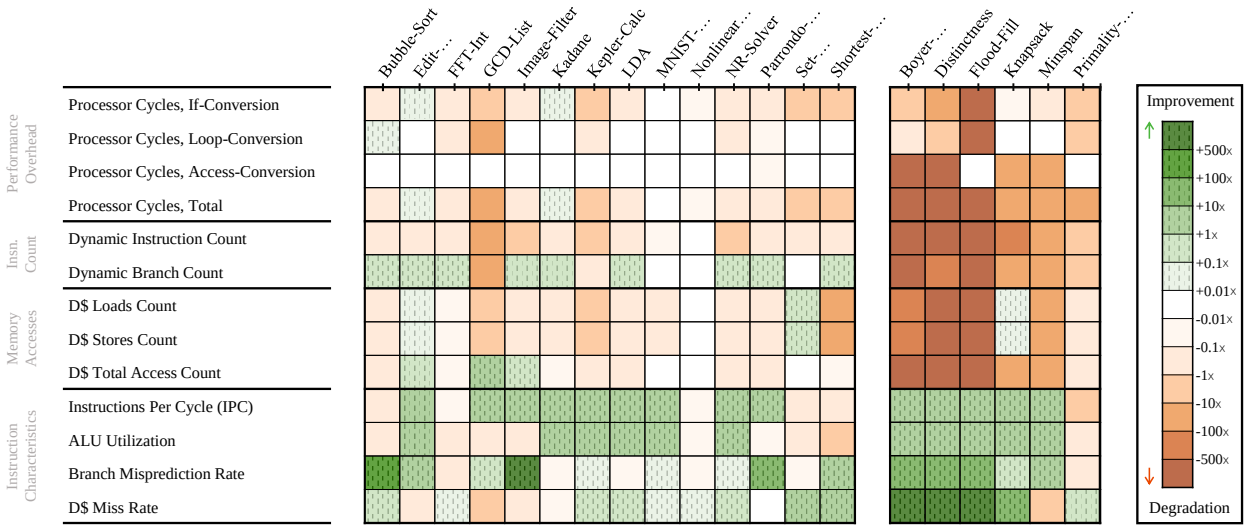


Figure 9.4: **Characterizing Trends in Oblivious Workloads.** This heatmap shows the trends in runtime performance and instruction characteristics across our twenty workloads. While dynamic instruction count consistently increases, branch count is varied, as if-conversion removes branch instructions, whereas other conversions add them. Instruction characteristics show improvement across multiple workloads.

load's IPC improves by 2.76×, from 0.71 to 1.97. Thus, the overall performance slowdown incurred by these changes is less than what is implied by the instruction overhead alone, at ∼1697×. Five workloads experience more than 2× IPC improvement (*Boyer-Moore-Search*, *Distinctness*, *Flood-Fill*, *Kepler-Calc*, and *Minspan*). These results demonstrate that the processor can execute more instructions in parallel and reduce stalling when running oblivious workload variants.

9.5 Algorithms for Data-Oblivious Execution

Workloads that incur higher asymptotic complexity resulting from the mechanized data-oblivious transformations suffer from significant runtime slowdowns (∼179× geomean). However, by leveraging our insights about sources of this overhead, we can redesign these applications' core algorithms to mitigate slowdowns. In this section, we show how algorithmic changes can be used to overcome the overheads of three workloads (*Boyer-Moore-Search*, *Distinctness*, and *Flood-Fill*). We also present a fourth case study on sorting (*Bubble-Sort*), which has been developed significantly by existing literature. Compared to mechanically applying data-oblivious transforms to native baseline programs, our algorithmic optimizations provide a geomean speedup of 86.4×, as listed in Table 9.4.

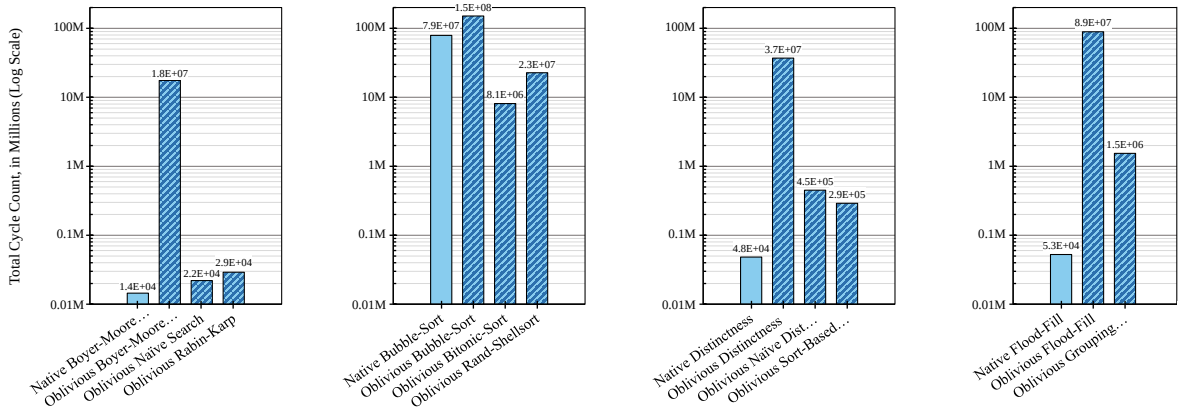
Benchmark	Oblivious Complexity	Input Size	Overhead over Native	Speedup
Boyer-Moore-Search	$O(n^2m)$	$n = 512, m = 8$	1214×	-
Naïve-Search	$O(nm)$	$n = 512, m = 8$	1.5×	796×
Rabin-Karp	$O(nm)$	$n = 512, m = 8$	2.0×	603×
Bubble-Sort	$O(n^2)$	$n = 4096$	1.9×	-
Bitonic-Sort	$O(nlog^2n)$	$n = 4096$	0.1×	18.6×
Rand-Shellsort	$O(nlogn)$	$n = 4096$	0.3×	6.7×
Distinctness	$O(n^3)$	$n = 128$	767×	-
Naïve Distinctness	$O(n^2)$	$n = 128$	9.3×	82.1×
Sort-Based Distinctness	$O(nlog^2n + n)$	$n = 128$	6.0×	127×
Flood-Fill	$O(n^4)$	$n = 32$	1697×	-
Grouping Flood-Fill	$O(n^2)$	$n = 32$	29.3×	58.0×
GEOMEAN				86.4×

Table 9.4: **Case Studies.** Above, we list the benchmark variants for our four case studies: string search, integer sort, element distinctness, and flood fill. Overall, our algorithmic improvements provide a ∼86× geomean speedup compared to the mechanically converted oblivious workload.

9.5.1 String Search

String search algorithms locate a word or pattern of length m within a text of length n. Native *Boyer-Moore-Search* has one of the fastest best-case complexities for pattern matching compared to other algorithms. This advantage stems from heuristics that accelerate the matching process. However, ***these heuristics are not amenable to data-oblivious transformations*** and result in a performance overhead of 1214×. In general, we find that heuristics often conflict with data-obliviousness, as they introduce data-dependent execution into workloads. We have often found that removing a heuristic is more efficient in an oblivious setting than attempting to transform it, as is the case with oblivious *Naïve-Search* (described below). Figure 9.5a shows the total cycle count of native *Boyer-Moore-Search* and these three oblivious workload variants.

Why is Oblivious Boyer-Moore slow? When searching for a pattern, the Boyer–Moore string-search algorithm is designed to skip over sections of text based on a heuristic. The VIP-Bench *Boyer-Moore-Search* implementation specifically uses the bad-character rule to shift the text anywhere from one character to the full length of the pattern (m). This optimization gives the native workload a best case runtime of $O(n/m)$ to locate a match. However, conversion of this heuristic severely degrades performance. Specifically, this input-dependent heuristic is used to dictate accesses to the text array. These array indexes are transformed via access-conversion and replaced with an ORAM search. In *Boyer-Moore-Search*, access-



(a) **String Search.** (b) **Integer Sort.** (c) **El. Distinctness.** (d) **Flood Fill.**

Figure 9.5: **Total Cycle Count for Optimized Workloads (Log Scale).** This figure shows the overall processor cycle count for the native, oblivious, and optimized workloads discussed in our case studies on a log scale. Solid bars represent the native (unmodified) workload, whereas dashed bars represent the oblivious workload variants. Our study demonstrates how algorithm redesign can significantly reduce data-oblivious programming overheads. For example, *String-Search*'s oblivious performance overhead is reduced from 1214× to under 2×, while *Distinctness* is reduced from 767× to under 10×. All relative performance overheads are listed in Table 9.4.

conversion is not only required once, but at multiple points throughout the algorithm. This transformation slows the program by 177× atop changes from loop- and if- conversion. Of all of our workloads, *Boyer-Moore-Search* also suffers from the highest overhead from if-conversion (4.7×) due to conditional predicates related to usage of the above heuristic.

9.5.1.1 Naïve-Search

Since much of the overheads of oblivious *Boyer-Moore-Search* are related to the usage of the bad-character heuristic, we remove it, reducing the algorithm to a naïve, iterative search. The oblivious *Naïve-Search* workload loops over every index of the text array, from one to n, and compares the subsequent m characters to check for a pattern match. This simple approach does not require access-conversion, making it significantly faster than oblivious *Boyer-Moore-Search*, with slowdowns under 1.5× for a 512-length text. Thus, *Naïve-Search* achieves a 796× speedup over the mechanically-converted *Boyer-Moore-Search*.

9.5.1.2 Rabin-Karp

Rabin-Karp is a string-search algorithm from the literature that uses a rolling hash to efficiently locate a pattern within a larger text string [270]. Specifically, *Rabin-Karp* slides over the text incrementally, checking if the hash values of the text and pattern in the current win-

dow are equivalent. The use of a hash function serves to reduce the amount of computation needed to compare the text and pattern strings. This algorithm's control-flow structure is similar to that of *Naïve-Search*, since both process every element of the text array. Thus, *Rabin-Karp* is amenable to data-oblivious programming.

We observe that oblivious *Naïve-Search* slightly outperforms *Rabin-Karp*, since the naïve comparison logic can be unrolled and parallelized. Specifically, *Naïve-Search* does more work than *Rabin-Karp* (1.7× more instructions), but this work is easier in nature (2.3× better IPC). Ultimately, both algorithms provide a two orders of magnitude speedup compared to the mechanically-converted oblivious code.

9.5.2 Integer Sort

The problem of sorting elements has been well studied in the literature. Many advanced sorting algorithms exist that can outperform our mechanically converted *Bubble-Sort* workload, as they have better time complexity and are innately data-oblivious. Specifically, we describe and evaluate two existing sorting schemes that are data-oblivious: Bitonic Sort and Randomized Shellsort. Figure 9.5b shows the total cycle count of native *Bubble-Sort* and these three oblivious workload variants.

Why is Oblivious Bubble-Sort slow? *Bubble-Sort* iteratively compares and conditionally swaps adjacent array elements. If-conversion transforms these compare-swap operations such that the adjacent array elements are overwritten on every iteration. This greatly increases the number of stores dispatched to the memory system and accounts for much of the performance overhead incurred by oblivious *Bubble-Sort*. However, obliviously performing a compare-swap operation will *always* induce additional stores. Thus, to minimize data-oblivious overheads, ***we must optimize the number of compare-swap operations performed by the algorithm***. This goal mirrors that of many native (non-oblivious) sorting algorithms.

9.5.2.1 Bitonic Sort

Bitonic-Sort [271] is a parallel algorithm for sorting that performs a fixed, input-independent sequence of compare-swap operations. *Bitonic-Sort* takes a divide-and-conquer approach by splitting the array into several sub-arrays that are alternating bitonic sequences (*i.e.*, sequences that are first increasing, then decreasing). The algorithm then repeatedly merges these sub-arrays into larger bitonic sequences until the entire array is sorted. The number of compare-swap operations in *Bitonic-Sort* is fixed and less than that of *Bubble-Sort*—*Bitonic-Sort* has a complexity of $O(nlog^2n)$. As a result, *Bitonic-Sort* does significantly less work

than oblivious *Bubble-Sort*, committing a tenth of the number of instructions, and is about 18.6× faster.

9.5.2.2 Randomized Shellsort

Randomized Shellsort (*Rand-Shellsort*) [272] is a randomized variant of the Shellsort algorithm that was designed to always run in $O(nlogn)$ time. To achieve this, *Rand-Shellsort* integrates randomness into the traditional sorting algorithm. *Rand-Shellsort* performs multiple passes over the input array, sometimes selecting pairs of elements deterministically (similar to the Shellsort algorithm), and other times selecting pairs of elements based on a random permutation. This pattern enables *Rand-Shellsort* to reduce the number of compare-exchange operations while sorting the array with high probability.

Interestingly, the control flow and memory access patterns of *Rand-Shellsort* are *not* known at compile time due to the random selection of compare-exchange operations. This is in sharp contrast to the other workloads examined, which approach obliviousness through static execution. Namely, *Rand-Shellsort* exemplifies an alternative approach to designing data-oblivious algorithms—leveraging statistical randomness instead of naïve (and often worst-case) control flow.

Rand-Shellsort offers a performance improvement of 6.7× when compared with oblivious *Bubble-Sort*. This speedup is not as great as *Bitonic-Sort*, as *Rand-Shellsort*'s randomized iteration over array regions results in more control-logic and a worse branch misprediction rate (4.3%). These added pipeline stalls cause *Rand-Shellsort* to have the worst IPC of all four sorting variants discussed in this section, at just under one instruction per cycle.

9.5.3 Element Distinctness

Distinctness checks whether all elements in an array are unique. There are many approaches to solve this problem. The algorithm provided by VIP-Bench [15] inserts all elements in the array into a binary search tree (BST) and early-exits when detecting a duplicate element. However, this early-exit and use of a BST is not amenable to data-oblivious programming. In this case study, we describe and evaluate two simple alternatives to computing element distinctness: oblivious *Naïve-Distinctness* and *Sort-Based Distinctness*. Figure 9.5c shows the total cycle count of native *Distinctness* and these three oblivious workload variants.

Why is Oblivious Distinctness slow? *Distinctness* checks element uniqueness by inserting values into a BST. This approach is not amenable for an oblivious setting because, *i)* The early-exit is removed during loop-conversion, requiring all elements to be inserted in the BST, and *ii)* The BST traversal is data-dependent and must be modified during access-conversion

to be oblivious. Recalling Figure 9.3, access-conversion introduces a large portion of the runtime overhead. This transformation, alongside loop-conversion, increases the complexity of *Distinctness* from $O(nlogn)$ to a whopping $O(n^3)$ for the oblivious variant.

9.5.3.1 Naïve Distinctness

To eliminate overheads associated with access-conversion, oblivious *Naïve-Distinctness* uses a static, nested for-loop to iteratively check if each element in the array is unique. Like oblivious *Distinctness*, the algorithm cannot early-exit after detecting a duplicate element. Thus, *Naïve-Distinctness*' iterative approach has a greater asymptotic complexity compared to the native (non-data-oblivious) variant, at $O(n^2)$. However, *Naïve-Distinctness* significantly reduces the number of branches ($\sim$40$\times$) and memory accesses (128$\times$) compared to the oblivious baseline, thus offering an 82$\times$ runtime speedup.

9.5.3.2 Sort-Based Distinctness

While *Naïve-Distinctness* serves to mitigate the access-conversion overheads incurred by oblivious *Distinctness*, its time complexity can be improved by leveraging other algorithms we have already discussed. Specifically, *Sort-Based-Distinctness* first sorts the array using oblivious *Bitonic-Sort*, and then iterates through the array to check if any adjacent elements are duplicates. Under this algorithm, we can compute element distinctness in $O(nlog^2n+n)$ time. *Sort-Based-Distinctness* provides a 127$\times$ speedup over the mechanically converted workload for n=128 data set sizes.

9.5.4 Flooding Cells with Flood-Fill

Flood-Fill iterates over a matrix to alter any cells that are "connected" to a chosen target cell. This functionality is used for the bucket-fill tool in software graphics programs (*e.g.,* Microsoft Paint), which colors all same-colored pixels connected to a selected pixel. As discussed in Section 9.4, the recursive nature of the native algorithm suffers from significant performance degradation when converted to data-oblivious (1697$\times$ for n=32). In this case study, we work to develop an alternative, iterative *Flood-Fill* implementation that reduces the number of passes needed to process adjacent cells. Figure 9.5d shows the total cycle count of native *Flood-Fill* and these two oblivious workload variants.

Why is Oblivious Flood-Fill slow? Translating *Flood-Fill* to be data-oblivious is non-trivial due to its complex control logic. When flooding a particular matrix cell, native *Flood-Fill* recurses on neighboring cells of the same color. Under mechanized loop-conversions (Section 9.2), oblivious *Flood-Fill* makes n^2 passes over the matrix to ensure that all connected

cells are flooded. However, this change increases the complexity significantly, from $O(n^2)$ to $O(n^4)$, since matrix cells are now visited more than once. Of the workload's 1697× performance slowdown, more than 759× originates from loop-conversion, whereas if-conversion only accounts for 2.2×. For these reasons, our optimized implementation serves to reduce the impacts from loop-conversion.

9.5.4.1 Grouping Flood-Fill

We developed a new, data-oblivious flood-fill algorithm that reduces the number of passes made over the $n \times n$ input matrix. *Grouping-Flood-Fill* makes a fixed number of matrix passes. First, the algorithm scans the matrix to initialize all indices to a particular group based on their same-colored adjacent array indices (*e.g.*, if `M[0][1]` is the same color as `M[0][0]`, it is assigned to the same group). This pass begins to coalesce same-colored elements into groups but does not capture all connected components. Second, the algorithm scans the matrix in forward then reverse order to propagate the groups to other connected indices of the same color. Groups are recorded using a bitmap, which allows us to efficiently combine two groups via the bitwise OR (`|`) operator. After these passes, all cells assigned to the same group (checked via the bitwise AND (`&`) operator) are connected. Each complete same-colored connected group will be coalesced. The final pass iterates over the matrix to recolor all cells that share the same group as the target cell, completing the flood fill.

The *Grouping-Flood-Fill* algorithm provides a 58× speedup over oblivious *Flood-Fill*. While this revised oblivious algorithm does not exhibit more predictable control flow, it does reduce the number of memory accesses, branches, and therefore branch mispredicts encountered at runtime. These changes offer a performance boost by reducing stalls in the pipeline.

9.6 Discussion and Limitations

The workloads in VIP-Bench [15] contain only a subset of common data structures—primarily fixed-sized arrays and matrices, with graphs represented as adjacency matrices. This is a limitation of our analysis since data structures can significantly affect how amenable an algorithm is to data-oblivious transformations. For example, native *Distinctness* used a binary search tree (BST). This choice of data structure incurred high overheads under mechanised conversions, especially when compared with the iterative array-based approach we adapted in Section 9.5. Other data structures not evaluated here, such as linked lists, may similarly incur high overheads. Studying how to represent sparse graphs and other data structures in data-oblivious programs is an interesting line of future work.

9.7 Related Work

Languages and compilers are promising mechanisms to assist in data-oblivious code transformations. However, as exhibited by Section 9.5, algorithmic development is also warranted, as simple automated compiler support alone will not achieve efficient data-oblivious programs. We detail existing work related to these two efforts below. We note that much of the work on data-oblivious languages is coupled with secure computation, which encrypts data values. While these efforts are critical, we believe it is also useful to study data-oblivious programs separately, as obliviousness can be beneficial beyond security applications. Further, decoupling data-obliviousness and secure computation promotes the development of unified interfaces and software frameworks for different primitives.

Oblivious Languages and Tools. Obliv-C [235] provides high-level abstractions that enable data-oblivious programming. Specifically, Obliv-C extends C with new oblivious data types and control structures (*i.e.*, conditionals, similar to CMOV). Type rules are enforced at compile time to detect non-oblivious operations. To further ensure insulation from information leaks, this security-oriented work also encrypts data values such that any non-oblivious use of them at runtime will result in errors or garbled values. ObliVM [234] provides a domain-specific language (DSL) for secure computation that enables developers to easily write data-oblivious programs that are compiled to use secure computation primitives like garbled circuits. ObliVM also exposes oblivious programming abstractions like ORAM to its users. Data-Oblivious ISA Extensions (OISA) [109] works towards extending data-oblivious guarantees to architectures, as hardware has been shown to employ data-dependent optimizations that violate obliviousness. Finally, we regard the verification of oblivious algorithms as critical to this domain. ObliCheck [273], which uses symbolic execution in Jalangi [274], is the only tool we know of that formally verifies data-oblivious algorithms.

Oblivious Algorithms. There is significant research on improving algorithm and data structure performance in oblivious contexts. Much of this development has occurred around oblivious search using oblivious RAM (ORAM) protocols. Advanced ORAM protocols enable secure secret-based addressing with reduced overheads. For example, Path ORAM [105] is a popular tree-based ORAM scheme that performs $O(logn)$ accesses per lookup (compared to n accesses for the trivial ORAM search we implement for access-conversion). Recent work has focused on further improving the costs of oblivious search (*e.g.*, by reducing disk seeks [115]), as well as extending these schemes to multi-party settings (*e.g.*, [116]).

Recent research has also worked to optimize algorithms for oblivious settings beyond ORAM. Wang et al. apply data-oblivious techniques to data structures (stacks, queues,

priority-queues, and maps/sets) and consider their use in graph and memory-allocation algorithms [275]. Most notably, this work implements and evaluates an oblivious maximum flow algorithm on sparse graphs. Blanton et al. present oblivious algorithms for classical graph applications on adjacency matrices (*i.e.*, breadth-first search, single-source single-destination shortest path, minimum spanning tree, and maximum flow) [276]. ObliDB presents efficient, oblivious query algorithms and implements them for general-purpose SQL databases on Intel SGX [228]. Oblix [265] combines oblivious search techniques with hardware enclaves to develop an efficient search index that scales to large databases.

9.8 Chapter Summary

In this work, we examined the performance of data-oblivious programs, a class of algorithms that have great potential in the arenas of security, branch and memory optimization, and hardware synthesis. We found that, while mechanically transforming a program to be data-oblivious can result in very high overheads, this is not the case for all workloads, particularly those whose asymptotic complexity is not impacted. Additionally, data-oblivious transformations often improved branch prediction, memory performance, and ALU utilization, helping offset the other costs of oblivious programming. Finally, we examined the value of redesigning applications' core algorithms for an oblivious model of computation. We found this approach particularly fruitful, reducing overheads by more than an order of magnitude, on average. These results are encouraging as they suggest that additional research on compiler and algorithmic optimizations can improve the efficiency of data-oblivious algorithms and, therefore, also Sequestered Encryption.

CHAPTER 10

Conclusion

Our approach to designing encrypted architectures has evolved over the course of this dissertation as we have learned lessons from our research. Overall, our work demonstrates the effectiveness of architectural approaches for security and how information asset protections can provide broad coverage against exploits. These themes span beyond protecting information assets and have greater implications for the future of hardware security research.

- ***Vulnerability-Agnostic Design:*** Software vulnerabilities are so pervasive that they are expected to be present in complex systems software. As systems grow, avoidance defenses (§2.4) that focus on patching vulnerabilities will become increasingly ineffective and non-durable. It is insurmountable to find all software vulnerabilities since this would require formal methods that do not scale with large systems. Architectural approaches enable us to design *vulnerability-agnostic* systems, as any software running atop the architecture, including programs that contain vulnerabilities, can be insulated from attack. This principle undercuts all of our designs.

- ***Hardware-Based Defenses:*** Architectural approaches lend themselves to efficient, vulnerability-agnostic designs and the comprehensive protection of information assets. However, pursing hardware-based defenses is challenging because security analysis is best done when attackers have access to a fully implemented and detailed system. Current approaches to computer architecture modeling do not accommodate this need, as hardware designs are proprietary and a majority of research is done using functional or cycle-accurate simulators (*e.g.*, gem5 [6]). The RISC-V open-source ecosystem has recently become a catalyst for hardware-based defenses as it allows for full system implementations. RISC-V enabled our Morpheus research team and the broader FETT program [34, 197] to deploy Amazon AWS F1 FPGA-cloud instances for security researchers to hack. This opportunity was critical to the evaluation of the Morpheus II architecture. We hope that future advancements to the RISC-V ecosystem empower more comprehensive security analysis and, ultimately, the broader adoption of hardware-based defenses.

- ***Embracing Strong Encryption:*** Our defense approaches evolved from using encryption minimally in memory, to using it comprehensively with reduced security, to embracing strong encryption in the pipeline despite its latencies. This progression was motivated by growing security concerns and the relinquishment of performance-first design. Encryption is a powerful tool to protect information assets as encrypted assets can be totally exposed to an attacker yet remain secret. By keeping values encrypted in the many leaky microarchitectural structures, we can guard them from attack (*e.g.*, §2.5.3, §2.5.4) without invasive changes to the architecture. However, strong encryption is costly. Time-tested cryptographic schemes like AES take forty cycles at a 2.5 GHz clock speed, compared to an add operation that takes a single cycle. Luckily, there exists many opportunities to leverage encryption while avoiding these latencies. In Chapter 8, we showed how architectural optimizations can be used to mitigate encryption overheads without imbuing timing side channels. This result serves as an inspiration for future work to safely optimize the performance of security defenses. It is also possible to accelerate the AES implementation itself with a more deliberate hardware design. Further, emerging lightweight ciphers (*e.g.*, K-Cipher [240]) or ciphers designed for hardware implementations (*e.g.*, QARMA [10]) have the potential to reduce slowdowns, especially as they are welcomed by the community.

- ***Toward Side-Channel Resilience:*** Our defense approaches also evolved to provide better protection from side channels. Our early work protecting code and pointers decrypted assets before use. Thus, information about encrypted assets were willfully leaked through side channels (*e.g.*, an encrypted code pointer is revealed when used to access the I-cache [101]). Re-keying served as a mechanism to nullify leaked information. However, holistic side-channel-resilient approaches offer greater security. With Sequestered Encryption [35], we embraced side channel leakage in our threat model and worked to explicitly design a hardware enclave that is resilient to timing analysis. Further, we were able to verify this claim formally because of SE's minimal design [261]. This work demonstrates how low-trust architectures can make significant progress towards side-channel resilience, especially as side channels have plagued commercial hardware protections in recent years [103, 118, 219, 220, 221, 277].

These lessons not only serve as takeaways from this dissertation but also inspire future work. For example, it is interesting to consider how timing-side-channel resilience can be adapted for Morpheus, which must perform tasks like indexing the I-cache with a protected pointer, or how SE can be designed to thwart other classes of side channels, such as power analysis. Below, we briefly summarize the work presented in this dissertation before discussing directions for future research.

10.1 Dissertation Summary

There are increasing incidences of security exploits and high-profile data breaches that leverage not only novel weaknesses (*e.g.*, Spectre [16] and Meltdown [17]) but also vulnerabilities that we have known about for decades yet remain widespread (*e.g.*, memory errors [142]). In this dissertation, we leveraged an understanding of the security landscape to disrupt this trend and design more durable solutions. In Chapter 2, we provided a framework for organizing vulnerability sources throughout the program life cycle and used this framework to categorize existing attacks and defenses. From this chapter, we recognized the repeated usage of *information assets* in security exploits across attack genealogies. This insight informed our approach on building security and privacy solutions—applying protections to a few critical information assets can potentially achieve broad coverage against security exploits. Further, by leveraging encryption and custom hardware extensions, these defense can become efficient and resilient to attack.

In this dissertation, we explored how architectures that encrypt information assets (*i.e.*, code, pointers, and data) can provide efficient and comprehensive security and privacy protections. In Chapter 3, we presented our first instantiation of an encrypted architecture that protects information assets: Morpheus [31]. The Morpheus secure architecture leverages an ensemble of moving target defenses to thwart control-flow attacks, layering address space displacement, code and pointer encryption, attack detection, and runtime re-keying (or "churn"). While Morpheus is effective and performant, our design choices prioritized runtime efficiency, resulting in two security flaws. First, Morpheus only encrypts pointer and code representations in memory, minimally protecting pointers in the pipeline. Second, Morpheus requires significant architectural modifications, including the usage of tagged memory. Our subsequent chapters iterated on the Morpheus design to address these deficiencies. In Chapter 4, Displaced and Dilated Address Spaces (DDAS) [32] expanded Morpheus' address space displacement protection to better protect pointers in the pipeline. Our evaluating demonstrated that DDAS can detect 99.996% of pointer manipulations with minimal performance impacts. Our research continued to evolve to leverage encryption more comprehensively. In Chapter 5, we revisited the original Morpheus design to leverage encryption in the pipeline while minimizing microarchitectural impacts. We presented Morpheus II [33], a secure architecture that processes encrypted code and code pointers, better protecting them from exfiltration. The strength of Morpheus II was demonstrated in a three-month bug-bounty program, where over five-hundred attackers were unable to exploit a vulnerable application running atop the secure architecture. This result demonstrated the strength of encrypted architectures for protecting information assets and exemplified vulnerability-agnostic design.

After the success of Morpheus II in thwarting control-flow attacks, we began to consider how encrypted architectures can be leveraged to protect other assets. In the second half of this dissertation, we turned our attention to protecting sensitive data to provide data privacy in the presence of software vulnerabilities. We presented Sequestered Encryption (SE) [35]—a hardware capability that provides comprehensive data privacy by removing all sensitive plaintext values from software-accessible architectural state. We designed the SE architecture and evaluated its performance when configured with three different cryptographic ciphers: AES, Simon, and QARMA. Due to long cipher latencies on the critical path, performance overheads were moderate, at an average 36× slowdown with AES. However, architectural optimizations and software changes can significantly decrease these overheads. In Chapter 8, we find that classic architecture techniques like caching and forwarding can almost entirely mask cipher latencies. With our optimized microarchitecture design, performance overheads were reduced to just 2.5× native computation, with programmatic changes accounting for more than half of this overhead. We concluded this dissertation by investigating the software overheads accounting for the majority of the remaining performance slowdowns. We presented a systematic characterization of the runtime overheads of encrypted data architectures and provide examples of ways programs can be designed to avoid these slowdowns.

10.2 Future Directions

I believe that encrypted architectures can revolutionize the fields of security and privacy. Many exciting directions for future work follow from the research presented in this dissertation. Some of these ideas are summarized below.

Secure Memory Accesses for Encrypted Architectures. The Morpheus architectures leak encrypted pointer values to cache side channels during memory lookups. To avoid this leakage, Sequestered Encryption requires costly programmatic conversions to transform data-dependent memory accesses. As shown in Chapter 9, these access conversion incur the largest performance slowdowns of all data-oblivious constructs. Oblivious RAMs (ORAMs) have the potential to address both of these issues. ORAMs are algorithms that randomize memory access to decouple them from the data being computed upon. By integrating ORAMs in architectures directly, programs can be insulated from memory side channels automatically. I-cache accesses with decrypted pointers in Morpheus would be secure, and access conversion in SE would not be needed. Further research is needed to understand and optimize the performance impacts of ORAM algorithms for real systems. ORAMs are

typically characterized by their asymptotic complexity in the number of additional memory accesses they require. This characterization gives a sense of the relative performance of competing ORAM schemes, but ignores many nuanced system aspects that significantly impact absolute runtimes (*e.g.*, how cache performance is impacted by the random-nature of ORAM accesses). Understanding absolute runtimes on real systems is critical to designing efficient implementations of ORAMs in hardware.

Powerful Extensions to Encrypted Data Architectures. Encrypted architectures enable capabilities beyond what is presented in this dissertation. Specifically, extensions to Sequestered Encryption can enable *functional encryption*—a class of cryptographic algorithms that permit third-parties to learn the plaintext result of a particular function rendered over some encrypted data, while upholding the confidentiality of that data (*e.g.*, a company can learn the average of a data set, but cannot directly see any of that data). This powerful mechanism enables data owners to place strict constraints on what other parties can do with their data (*e.g.*, the company cannot learn the median). However, cryptographic approaches to functional encryption are largely theoretical. With hardware support, we can implement this functional encryption securely, potentially having far-reaching impacts on data privacy in commercial settings.

Software Support for Encrypted Architectures. Sequestered Encryption requires significant software changes to denote sensitive data and accommodate data-oblivious programming. We left Chapter 9 understanding the performance impacts of data-oblivious changes and the significance of algorithm selection. However, much more work can be done towards designing efficient data-oblivious algorithms and aiding programmers in using these frameworks. Software support for encrypted computation must be developed in tandem with secure architectures, as making privacy solutions accessible to programmers is integral to their adoption.[1] Compilers can assist in making privacy technologies more accessible to programmers by automating the mapping of program constructs onto privacy technologies and performing other optimizations. Our findings in Chapter 9 can be generalized into data-oblivious compilation flows in future work.

More broadly, it is interesting to consider how we can protect legacy code with vulnerability-agnostic architectures. Morpheus and SE can run legacy code, as programs can execute atop the architecture without engaging any of its protections. However, these

[1]Such importance is evidenced by homomorphic encryption. While extremely powerful, currently available homomorphic encryption libraries expose cryptographic primitives to users (*e.g.*, batching, relinearization, and other ciphertext maintenance operations), making these technologies difficult for non-experts to use.

architectures cannot protect legacy code. Programs must be recompiled to use the defense's ISA extension (in the case of Morpheus II [33] and SE [35]) or, at minimum, be screened for metadata. Techniques like Cryptographic Capability Computing (C^3) [27] support legacy programs by intercepting targeted instructions at runtime. Future research could consider how to integrate this approach for the defenses presented in this dissertation.

Returning to Security. While we concluded this dissertation focusing on data privacy, much more work is needed in the security space. Our review of attack genealogies in Chapter 2 showed how exploits largely rely on knowledge of key information assets, like code and pointers. Morpheus II [33] encrypts code and code pointers to thwart control-flow attacks. While it withstood red-teaming efforts in the DARPA FETT bug-bounty program, its design was imperfect as it did not prevent information assets from leaking through side channels. Insulating this final attack vector has the potential to stop control-flow exploits indefinitely. More broadly, additional work can be done to protect other asset domains, such as data pointers, to stop even broader swaths of the attack surface. The lessons we have learned from Sequestered Encryption can inform the design of even more powerful security architectures. Yet, as we return to security, we must also expend more effort on reducing performance overheads, as the community is more sensitive to slowdowns. Our later chapters demonstrate how we can approach efficient designs without compromising security.

BIBLIOGRAPHY

[1] Lauren Biernacki, Mark Gallagher, Zhixing Xu, Misiker Tadesse Aga, Austin Harris, Shijia Wei, Mohit Tiwari, Baris Kasikci, Sharad Malik, and Todd Austin. Software-Driven Security Attacks: From Vulnerability Sources to Durable Hardware Defenses. *ACM Journal on Emerging Technologies in Computing Systems*, 17(3), 2021.

[2] Lauren Biernacki, Austin Harris, Tarunesh Verma, Sharad Malik, and Todd Austin. Encrypted Architectures for Protecting System Information Assets: The Morpheus Case Study. In *Communications of the ACM (CACM)*. Under Submission.

[3] Chilik Tamir. WhatsApp Buffer Overflow Vulnerability: Under the Scope. *Zimperium*, 2019.

[4] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[5] PaX Team. PaX Address Space Layout Randomization (ASLR). Online. Available: http://pax.grsecurity.net/docs/aslr.txt, 2003.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.

[7] Jack Kang. Freedom U500. Online. Available: https://content.riscv.org/wp-content/uploads/2017/12/Tue1224-SiFive_Freedom_U500-Kang.pdf, 2017.

[8] SiFive, Inc. SiFive S51. Online. Available: https://www.sifive.com/cores/s51, 2019.

[9] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization*, 14(2):14:1–14:25, 2017.

[10] Roberto Avanzi. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-

Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, 2017.

[11] Aaron Stillmaker and Bevan Baas. Scaling eauations for the Accurate Prediction of CMOS Device Performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

[12] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 367–382, 2016.

[13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. Implementation and Performance of the Simon and Speck Lightweight Block Ciphers on ASICs. *Unpublished work.*

[14] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 375–389, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Galane Basha Namomsa, Plato Gebremedhin, Fitsum Assamnew Andargie, Brandon Reagen, and Todd Austin. VIP-Bench: A Benchmark Suite for Evaluating Privacy-Enhanced Computation Frameworks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 139–149, 2021.

[16] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[17] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, 2018. USENIX Association.

[18] Nicole Perlroth and Ronen Bergman. WhatsApp Rushes to Fix Security Flaw Exposed in Hacking of Lawyer's Phone. *New York Times*, 2019.

[19] Swati Khandelwal. Hackers Used WhatsApp 0-Day Flaw to Secretly Install Spyware On Phones. *The Hacker News*, 2019.

[20] Swati Khandelwal. Facebook Sues Israeli NSO Spyware Firm For Hacking WhatsApp Users. *The Hacker News*, 2019.

[21] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. Online. Available: https://www.youtube.com/watch?v=PjbGojjnBZQ, 2019.

[22] Thomas F Dullien. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[23] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[24] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.

[25] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. *NDSS (Feb. 2017)*, 17:13, 2017.

[26] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 40:1–40:13, Piscataway, NJ, USA, 2016. IEEE Press, IEEE Press.

[27] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, et al. Cryptographic Capability Computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 253–267, 2021.

[28] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium*, volume 12, pages 91–104, Berkeley, CA, USA, 2003. USENIX Association.

[29] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[30] Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and X. Sean Wang. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer-Verlag, New York, NY, USA, 1st edition, 2011.

[31] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 469–484, New York, NY, USA, 2019. ACM.

[32] Lauren Biernacki, Mark Gallagher, Valeria Bertacco, and Todd Austin. Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 57–68, 2020.

[33] Austin Harris, Tarunesh Verma, Shijia Wei, Lauren Biernacki, Alex Kisil, Misiker Tadesse Aga, Valeria Bertacco, Baris Kasikci, Mohit Tiwari, and Todd Austin. Morpheus II: : A RISC-V Security Extension for Protecting Vulnerable Software and Hardware. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2021.

[34] DARPA. FETT Bug Bounty Helps Strengthen SSITH Hardware Defenses. Online. Available: https://www.darpa.mil/news-events/2020-01-28, 2020.

[35] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Fitsum Assamnew Andargie, and Todd Austin. Sequestered Encryption: A Hardware Technique for Comprehensive Data Privacy. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 73–84, 2022.

[36] Vijay D'Silva, Mathias Payer, and Dawn Song. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*, pages 73–87. IEEE, 2015.

[37] Alexander Peslyak (Solar Designer). Bugtraq: Getting Around Non-executable Stack (And Fix). Online. Available: https://seclists.org/bugtraq/1997/Aug/63, 1997.

[38] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A Complete Formal Semantics of X86-64 User-Level Instruction Set Architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1133–1148, 2019.

[39] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[41] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation. *USENIX; login*, 36(6), 2011.

[42] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. The Page-Fault Weird Machine: Lessons in Instruction-Less Computation. In *7th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2013.

[43] Rebecca Shapiro, Sergey Bratus, and Sean W Smith. "weird machines" in elf: A spotlight on the underappreciated metadata. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, 2013.

[44] Julien Vanegue. The Weird Machines in Proof-Carrying Code. In *2014 IEEE Security and Privacy Workshops*, pages 209–213. IEEE, 2014.

[45] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined Behavior: What happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, page 9. ACM, 2012.

[46] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. Taming Undefined Behavior in LLVM. *ACM SIGPLAN Notices*, 52(6):633–647, 2017.

[47] John Regehr. A Guide to Undefined Behaviour in C and C++. Online. Available: https://blog.regehr.org/archives/213, 2010.

[48] Hans-Juergen Boehm. How to Miscompile Programs with "Benign" Data Races. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*, Berkeley, CA, 2011. USENIX Association.

[49] Jennifer Paykin, Eric Mertens, Mark Tullsen, Luke Maurer, Benoît Razet, Alexander Bakst, and Scott Moore. Weird Machines as Insecure Compilation. *arXiv preprint arXiv:1911.00157*, 2019.

[50] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog Malicious Hardware. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 18–37. IEEE, 2016.

[51] Gennadiy Shvets. Enhanced Virus Protection / Execute Disable Bit. Online. Available: https://www.cpu-world.com/Glossary/E/EVP_XD.html, 2018.

[52] The MITRE Corporation. CVE-2001-0144: SSH CRC-32 Compensation Attack Detector Vulnerability. Online. Available: cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144, 2001.

[53] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE, IEEE Computer Society.

[54] Aleph One. Smashing the Stack for Fun and Profit. Online. Available: http://phrack.org/issues/49/14.html, 1996.

[55] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, 2012.

[56] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[57] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[58] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the Network and Distributed System Security Symposium 2016*, NDSS '16, Reston, VA, USA, 2016. Internet Society.

[59] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-Demand Live Randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 50–61, New York, NY, USA, 2016. ACM.

[60] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[61] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[62] M. Theodorides and D. Wagner. Breaking Active-Set Backward-Edge CFI. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 85–89, 2017.

[63] Polyverse Corporation. Polymorphing For Linux. Online. Available: https://www.dlt.com/sites/default/files/resource-attachments/2020-03/Polymorphing_for_Linux_Product_Brief-03.09.20.pdf, 2020.

[64] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[65] w00w00. w00w00 on Heap Overflows. Online. Available: https://www.cgsecurity.org/exploit/heaptut.txt, 1999.

[66] Shashank Gonchigar. ANI vulnerability: History repeats. *SANS Institute Information Security Reading Room*, 2007.

[67] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM, Association for Computing Machinery.

[68] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, 2014. USENIX Association.

[69] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[70] Alexander Peslyak (Solar Designer). Linux Kernel Patch from the Openwall Project: README. Online. Available: https://www.openwall.com/linux/README.shtml, 1997.

[71] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[72] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 281–289, New York, NY, USA, 2003. ACM.

[73] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[74] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C., 2013. USENIX.

[75] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proceedings of the Network and Distributed System Security Symposium 2014*, NDSS '14, Reston, VA, USA, 2014. Internet Society.

[76] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, USA, 2014. USENIX Association.

[77] Kanad Sinha and Simha Sethumadhavan. Practical Memory Safety with REST. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, ISCA '18, pages 600–611. IEEE Press, 2018.

[78] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348. IEEE, 2006.

[79] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd My Gadgets Go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585, 2012.

[80] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615, 2012.

[81] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. ASIST: Architectural Support for Instruction Set Randomization. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 981–992, New York, NY, USA, 2013. ACM.

[82] Kanad Sinha, Vasileios P. Kemerlis, and Simha Sethumadhavan. Reviving Instruction Set Randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28, Piscataway, NJ, USA, 2017. IEEE Press.

[83] John Kennedy and Michael Satran. Control Flow Guard for Windows Applications, Microsoft Docs. Online. Available: https://docs.microsoft.com/en-us/windows/desktop/secbp/control-flow-guard, 2018.

[84] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: Hardware-assisted Flow Integrity Extension. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 74:1–74:6, New York, NY, USA, 2015. ACM.

[85] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 941–951, New York, NY, USA, 2015. ACM.

[86] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913. ACM, 2015.

[87] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.

[88] William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. Getting in Control of Your Control Flow with Control-Data Isolation. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 79–90, 2015.

[89] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications security*, pages 298–307. ACM, 2004.

[90] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, 2014.

[91] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *NDSS*. The Internet Society, 2015.

[92] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[93] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 763–780. IEEE, 2015.

[94] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *NDSS*, 2016.

[95] Misiker Tadesse Aga and Todd Austin. Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 26–36, 2019.

[96] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168, Seoul, Korea, 2005. Springer, Springer-Verlag.

[97] Daniel J Bernstein. Cache-Timing Attacks on AES. 2005.

[98] Colin Percival. Cache Missing for Fun and Profit. In *Proceedings of the Technical BSD Conference (BSDCan) 2005*, 2005.

[99] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[100] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys via Branch Prediction. In *Topics in Cryptology – CT-RSA 2007*, pages 225–242. Springer, 2006.

[101] Onur Acıiçmez. Yet Another Microarchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 11–18, New York, NY, USA, 2007. ACM.

[102] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space Aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[103] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 663–680. USENIX Association, 2021.

[104] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, San Diego, California, USA, 2007. Association for Computing Machinery.

[105] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[106] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From Concept to Reality in the Intel®Xeon®Processor E5-2600 V3 Product Family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668. IEEE, 2016.

[107] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO-51, pages 974–987, Fukuoka, Japan, 2018. IEEE, IEEE Computer Society.

[108] Moinuddin K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 775–787, Fukuoka, Japan, 2018. IEEE, IEEE Press.

[109] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[110] Moinuddin K. Qureshi. New Attacks and Defense for Encrypted-Address Cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 360–371, Phoenix, Arizona, 2019. Association for Computing Machinery.

[111] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[112] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based cache attacks with a practical Fully-Associative design. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1379–1396. USENIX Association, 2021.

[113] Peter W. Deutsch, Yuheng Yang, Thomas Bourgeat, Jules Drean, Joel S. Emer, and Mengjia Yan. DAGguise: Mitigating Memory Timing Side Channels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 329–343, New York, NY, USA, 2022. Association for Computing Machinery.

[114] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 533–544, New York, NY, USA, 1998. ACM.

[115] Anrin Chakraborti, Adam J. Aviv, Seung Geol Choi, Travis Mayberry, Daniel S. Roche, and Radu Sion. rORAM: Efficient Range ORAM with O(log2 N) Locality. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[116] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-Party ORAM for Secure Computation. In *Proceedings, Part I, of the 21st International Conference on Advances in Cryptology – ASIACRYPT 2015*, page 360–385, Berlin, Heidelberg, 2015. Springer-Verlag.

[117] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv:1811.05441 [cs]*, pages 249–266, 2018.

[118] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[119] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-Flight Data Load. In *S&P*, 2019.

[120] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 769–784, 2019.

[121] Retpoline: A Software Construct for Preventing Branch-Target-Injection. Online. Available: https://support.google.com/faqs/answer/7625886.

[122] Retpoline: A Branch Target Injection Mitigation. Online. Available: https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.

[123] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[124] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 73–86, Columbus, OH, USA, 2019. ACM, Association for Computing Machinery.

[125] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 572–586, Columbus, OH, USA, 2019. ACM, Association for Computing Machinery.

[126] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 42–56, Columbus, OH, USA, 2019. Association for Computing Machinery.

[127] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757 [cs]*, 2018.

[128] Giorgi Maisuradze and Christian Rossow. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2109–2122, New York, NY, USA, 2018. ACM.

[129] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, WOOT'18, pages 3–3, Baltimore, MD, 2018. USENIX Association.

[130] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[131] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State Using Microarchitectural Side-Channels. *arXiv:1806.07480 [cs]*, 2018.

[132] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 954–968, Columbus, OH, USA, 2019. ACM, Association for Computing Machinery.

[133] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown With Leakage-Free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[134] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 723–735, Phoenix, Arizona, 2019. ACM, Association for Computing Machinery.

[135] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.*

[136] The Clang Team. Undefined Behavior Sanitizer (UBSan). Online. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[137] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX.

[138] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

[139] CVEs by Type. CVE Details: Vulnerabilities By Type. Online. Available: https://www.cvedetails.com/vulnerabilities-by-types.php, 2019.

[140] Gregory J. Duck and Roland H. C. Yap. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. ACM.

[141] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[142] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[143] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, 2015. ACM.

[144] lowRISC Project. lowRISC. Online. Available: https://lowrisc.org, 2018.

[145] Gregory T. Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. The Dover Inherently Secure Processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–5, 2017.

[146] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[147] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–86, Washington, DC, USA, 2004. IEEE Computer Society.

[148] Alec Roelke and Mircea R. Stan. RISC5: Implementing the RISC-V ISA in gem5. In *Proceedings of Computer Architecture Research in RISC-V*, CARRV '17, 2017.

[149] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[150] Matthew R. Guthaus, Jeff S. Ringenberg, Daniel Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, page 3–14, USA, 2001. IEEE Computer Society.

[151] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2006 Benchmarks. Online. Available: http://www.spec.org/cpu2006/CINT2006.

[152] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[153] AndesTech. Andes Technology RISC-V LLVM (GitHub). Online. Available: https://github.com/andestech/riscv-llvm, 2017.

[154] rv8. RISC-V musl libc. Online. Available: https://github.com/rv8-io/musl-riscv, 2018.

[155] James Yonan. OpenVPN 2.0.x man pages. Online. Available: https://openvpn.net/index.php/open-source/documentation/manuals/65-openvpn-20x-manpage.html, 2010.

[156] CWE-122: Heap-based Buffer Overflow. Online. Available: https://cwe.mitre.org/data/definitions/122.html, 2018.

[157] Yan Huang. Heap Overflows and Double-Free Attacks. Online. Available: http://homes.soic.indiana.edu/yh33/Teaching/I433-2016/lec13-HeapAttacks.pdf, 2016.

[158] riq and gera. Advances in Format String Exploiting. Online. Available: http://phrack.org/issues/59/7.html, 2001.

[159] OWASP. Format String Attack. Online. Available: https://www.owasp.org/index.php/Format_string_attack, 2015.

[160] Blexim. Basic Integer Overflows. Online. Available: http://phrack.org/issues/60/10.html, 2002.

[161] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 41–50, New York, NY, USA, 2011. ACM.

[162] Draper Laboratory. RIPE: The Runtime Intrusion Prevention Evaluator for RISC-V. Online. Available: https://github.com/draperlaboratory/hope-RIPE, 2018.

[163] Intel Corporation. Control-flow Enforcement Technology Preview. Online. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017.

[164] Aditya Venkataraman and Kishore Kumar Jagadeesha. Evaluation of Inter-Process Communication Mechanisms. Technical report, University of Wisconsin-Madison, Madison, WI, USA, 2015.

[165] Vince Bridgers. Real Time Linux Scheduling Comparison. In *Embedded Linux Conference 2015 (ELC 15)*, CA, 2015. The Linux Foundation.

[166] David Wagner and Paolo Soto. Mimicry Attacks on Host-based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[167] Mathias Payer. Too much PIE is bad for performance. Technical report, ETH Zurich, Department of Computer Science, Zürich, 2012. Technical Reports D-INFK.

[168] Fermin J. Serna. The info leak era on software exploitation. In *Black Hat USA*, 2012.

[169] N. Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 209–220, 2004.

[170] John Criswell and Vikram Adve. Chaos for a Fast, Secure, and Predictable Future. In *Fun and Interesting Thoughts at the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, FIT PLDI '10, 2010.

[171] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 2–12, New York, NY, USA, 2006. ACM.

[172] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. Data Randomization. Technical report, Microsoft Research, Redmond, WA, USA, 2008. MSR-TR-2008-120.

[173] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[174] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[175] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium*, USENIX-SS'06, USA, 2006. USENIX Association.

[176] Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. Security through Redundant Data Diversity. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 187–196, 2008.

[177] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium 2005*, NDSS '05, Reston, VA, USA, 2005. Internet Society.

[178] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[179] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium 2011*, NDSS '11, Reston, VA, USA, 2011. Internet Society.

[180] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, 2014.

[181] Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. Quantifying the Potential of Program Analysis Peripherals. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 53–63, Washington, DC, USA, 2009. IEEE Computer Society.

[182] Joseph L. Greathouse, Ilya Wagner, David A. Ramos, Gautam Bhatnagar, Todd Austin, Valeria Bertacco, and Seth Pettie. Testudo: Heavyweight Security Analysis via Statistical Sampling. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 117–128, Washington, DC, USA, 2008. IEEE Computer Society.

[183] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *USENIX Security Symposium*, pages 17–17, 2005.

[184] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for X86 and ARM. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and Communications security*, pages 299–310. ACM, 2013.

[185] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. Compiler-Generated Software Diversity. In *Moving Target Defense*, pages 77–98. Springer, 2011.

[186] William Hawkins, Anh Nguyen-Tuong, Jason D Hiser, Michele Co, and Jack W Davidson. Mixr: Flexible Runtime Rerandomization for Binaries. In *Proceedings of the 2017 Workshop on Moving Target Defense*, pages 27–37. ACM, 2017.

[187] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-Grained Address Space Randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490, 2012.

[188] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-Randomizing Instruction Addresses of Legacy X86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications security*, pages 157–168. ACM, 2012.

[189] Haizhi Xu and Steve J Chapin. Improving Address Space Randomization With a Dynamic Offset Randomization Technique. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 384–391. ACM, 2006.

[190] Xi Chen, Herbert Bos, and Cristiano Giuffrida. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 514–529. IEEE, 2017.

[191] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291. IEEE, 2014.

[192] RISC-V Foundation, Berkeley, CA, USA. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*, 2017. Editors Andrew Waterman and Krste Asanović. May 2017.

[193] ISO/IEC 9899:201x. Programming Languages: C. Standard, International Organization for Standardization, Geneva, CH, 2011.

[194] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[195] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882. ACM, 2018.

[196] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[197] DARPA. DARPA FETT Bug Bounty Program. Online. Available: https://fett.darpa.mil, 2020.

[198] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[199] David Weston. Meet the Microsoft Pluton Processor: The security chip designed for the future of Windows PCs. Online. Available: https://www.microsoft.com/security/blog/2020/11/17/meet-the-microsoft-pluton-processor/, 2020.

[200] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptol. ePrint Arch.*, 2013.

[201] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. SIMON and SPECK: Block Ciphers for the Internet of Things. *IACR Cryptol. ePrint Arch.*, 2015.

[202] Richard Barry. FreeRTOS. Online. Available: https://freertos.org, 2008.

[203] Philipp Klaus Krause. Stdcbench: A Benchmark for Small Systems. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, SCOPES '18, page 43–46, New York, NY, USA, 2018. Association for Computing Machinery.

[204] Giorgi Maisuradze, Michael Backes, and Christian Rossow. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 139–156, Austin, TX, 2016. USENIX Association.

[205] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[206] DARPA. System Security Integration Through Hardware and Firmware (SSITH). Online. Available: https://www.darpa.mil/program/ssith, 2017.

[207] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 29–42, Los Angeles, California, 2018. IEEE Press.

[208] Samuel Moore. Morpheus Turns a CPU Into a Rubik's Cube to Defeat Hackers. *IEEE Spectrum*, 2021.

[209] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[210] Mohamed Tarek Ibn Ziad, Miguel A Arroyo, Evgeny Manzhosov, Vasileios P Kemerlis, and Simha Sethumadhavan. Using Name Confusion to Enhance Security. *arXiv preprint arXiv:1911.02038*, 2019.

[211] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan. ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks.

In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 999–1012, 2021.

[212] Qualcomm Product Security. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions. *Qualcomm Technologies, Inc.*, 2017.

[213] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. HardBound: Architectural Support for Spatial Safety of the C Programming Language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[214] Gabe Cherry. DARPA Pitted 500+ Hackers Against this Computer Chip. The Chip Won. Online. Available: https://cse.engin.umich.edu/stories/morpheus-vs-everybody, 2021.

[215] Ariana Taylor. Did UM team build an unhackable computer chip? Nobody has beaten it, 2021.

[216] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[217] IBM Security. Cost of a Data Breach 2020. Online. Available: https://www.ibm.com/security/data-breach, 2020.

[218] Matthias Schunter. Intel Software Guard Extensions: Introduction and Open Research Challenges. In *Proceedings of the 2016 ACM Workshop on Software Protection (SPRO '16)*. Association for Computing Machinery, 2016.

[219] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in Darkness: Return-Oriented Programming Against Secure Enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, 2017.

[220] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, 2018.

[221] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[222] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the Seventh ACM workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM.

[223] Tiago Alves. Trustzone: Integrated Hardware and Software Security. *Information Quarterly*, 3:18–24, 2004.

[224] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *ACM Computing Surveys (CSUR)*, 51(4):1–35, 2018.

[225] Zhiniang Peng. Danger of Using Fully Homomorphic Encryption: A Look At Microsoft SEAL. *arXiv preprint arXiv:1906.07127*, 2019.

[226] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *Proceedings of the 12th International Conference on Information Security and Cryptology*, ICISC'09, pages 176–192, Berlin, Heidelberg, 2010. Springer-Verlag.

[227] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association.

[228] Saba Eskandarian and Matei Zaharia. An Oblivious General-Purpose SQL Database for the Cloud. *CoRR, abs/1710.00458*, 6:93–94, 2017.

[229] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *IACR Cryptol. ePrint Arch.*, page 549, 2017.

[230] Nancy Sumrall and Manny Novoa. Trusted Computing Group (TCG) and the TPM 1.2 Secification. In *Intel Developer Forum*, volume 32, 2003.

[231] Apple Inc. Storing Keys in the Secure Enclave. Online. Available: https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_secure_enclave.

[232] Kevin C Gotze, Gregory M Iovino, and Jiangtao Li. Secure Provisioning of Secret Keys During Integrated Circuit Manufacturing, 2017. US Patent 9,742,563.

[233] Kevin C Gotze, Jiangtao Li, and Gregory M Iovino. Fuse Attestation to Secure the Provisioning of Secret Keys During Integrated Circuit Manufacturing, 2014. US Patent 8,885,819.

[234] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 359–376. IEEE, 2015.

[235] Samee Zahur and David Evans. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.*, 2015:1153, 2015.

[236] Christian Stigen Larsen. A fast Mersenne Twister in C++ (GitHub). Online. Available: https://github.com/m0n0ph1/mersenne-twister-1, 2012.

[237] Intel Corporation. TinyCrypt Cryptographic Library (GitHub). Online. Available: https://github.com/intel/tinycrypt, 2017.

[238] McCoy, Calvin. Simon_Speck_Ciphers (Github). Online. Available: https://github.com/inmcm/Simon_Speck_Ciphers, 2018.

[239] Phantom1003. QARMA64 (Github). Online. Available: https://github.com/Phantom1003/QARMA64, 2019.

[240] Michael Kounavis, Sergej Deutsch, Santosh Ghosh, and David Durham. K-Cipher: A Low Latency, Bit Length Parameterizable Cipher. In *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020.

[241] Rui Zong and Xiaoyang Dong. Meet-in-the-Middle Attack on QARMA Block Cipher. *IACR Cryptol. ePrint Arch.*, page 1160, 2016.

[242] Robert G Brown. The DieHarder Test Suite. Online. Available: https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[243] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Technical report, Booz Allen & Hamilton, 2001.

[244] Microsoft SEAL (Release 3.6). Online. Available: https://github.com/Microsoft/SEAL, 2020. Microsoft Research, Redmond, WA.

[245] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[246] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic Computation of Edit Distance. In *International Conference on Financial Cryptography and Data Security*, pages 194–212. Springer, 2015.

[247] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 26–39. IEEE, 2021.

[248] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: An Architecture for Computing on Encrypted Data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1295–1309, 2020.

[249] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–62, 2021.

[250] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, 2021.

[251] Eduardo Chielle, Nektarios Georgios Tsoutsos, Oleg Mazonka, and Michail Maniatakos. Encrypt-Everything-Everywhere: ISA Extensions for Private Computation. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[252] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *ACM Sigplan Notices*, 35(11):168–177, 2000.

[253] David Champagne and Ruby B Lee. Scalable Architectural Support for Trusted Software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[254] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368. ACM, 2003.

[255] G Edward Suh, Charles W O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *32nd International Symposium on Computer Architecture (ISCA)*, ISCA '05, pages 25–36, Washington, DC, USA, 2005. IEEE, IEEE Computer Society.

[256] Ling Ren, Christopher W Fletcher, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Design and Implementation of the Ascend Secure Processor. volume 16, pages 204–216. IEEE, 2017.

[257] Naiwei Liu, Meng Yu, Wanyu Zang, and Ravi Sandhu. On the Cost-Effectiveness of TrustZone Defense on ARM Platform. In *Information Security Applications*, pages 203–214. Springer International Publishing, 2020.

[258] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[259] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining Lost Cycles With Hotcalls: A Fast Interface for Sgx Secure Enclaves. *ACM SIGARCH Computer Architecture News*, 45(2):81–93, 2017.

[260] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, 2016. USENIX Association.

[261] Qinhan Tan, Shibo Chen, Yonathan Fisseha, Lauren Biernacki, Jean-Baptiste Jeannin, Sharad Malik, and Todd Austin. Security Verification of Low-Trust Architectures. In *Proceedings of the 30th ACM Conference on Computer and Communications Security*, CCS '23. To Appear.

[262] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistantCPUs. *arXiv preprint arXiv:1905.05725*, 2019.

[263] Lauren Biernacki, Biniyam Mengist Tiruye, Meron Zerihun Demissie, Fitsum Assamnew Andargie, Brandon Reagen, and Todd Austin. Exploring the Efficiency of Data-Oblivious Programs. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 189–200. IEEE, 2023.

[264] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, 2017.

[265] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 279–296. IEEE, 2018.

[266] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[267] Le Xu. Securing the Enterprise with Intel® AES-NI. 2010.

[268] Trivikram Reddy, Krithiga Murugavel, and Jason Lowe-Power. gem5 Configuration for Intel's Skylake Micro-architecture. Online. Available: https://github.com/darchr/gem5-skylake-config.

[269] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 177–189, New York, NY, USA, 1983. Association for Computing Machinery.

[270] Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[271] K. E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[272] Michael T. Goodrich. Randomized Shellsort: A Simple Data-Oblivious Sorting Algorithm. *J. ACM*, 58(6), 2011.

[273] Jeongseok Son, Griffin Prechter, Rishabh Poddar, Raluca Ada Popa, and Koushik Sen. ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[274] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[275] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226, 2014.

[276] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[277] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devics. *Cryptology ePrint Archive*, 2016. https://eprint.iacr.org/2016/980.