Towards Defending Against Malware Attacks

by

Abdul Rahman Abu Elkhail

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical, Electronics, and Computer Engineering)
in the University of Michigan-Dearborn
2023

Doctoral Committee:

Professor Hafiz Malik, Chair
Assistant Professor Anys Bacha, Co-chair
Assistant Professor Mohamed Abouelenien
Assistant Professor Junaid Farooq
Assistant Professor Alireza Mohammadi

© Abdul Rahman Abu Elkhail 2023

All Rights Reserved

ORCID iD 0000-0002-6199-6468

For all the people

ACKNOWLEDGEMENTS

I would like to take this opportunity to convey my heartfelt thanks and profound appreciation to all those who have provided support and made significant contributions throughout the Ph.D. journey.

First and foremost, I would like to express my deepest gratitude and appreciation to my advisors, Dr. Hafiz Malik and Dr. Anys Bacha, for their invaluable guidance, support, and mentorship throughout the Ph.D. journey. Their expertise, encouragement, and unwavering commitment to my academic and personal growth have been instrumental in shaping the outcome of this work. I am truly grateful for their patience, insightful feedback, and dedication to helping me to complete this work.

I would also like to extend my heartfelt thanks to my family for their unwavering love, encouragement, and belief in my abilities. Their constant support, understanding, and motivation have been the pillars that have upheld me during challenging times. I am grateful for their sacrifices, understanding, and unwavering belief in me, which have provided me with the strength to pursue my academic goals.

Lastly, I would like to acknowledge and express my appreciation to the University of Michigan-Dearborn for creating an enriching academic environment. Their unwavering commitment to providing a nurturing research environment has empowered me to complete this work. I am grateful for the resources, facilities, and opportunities provided by the university, which have facilitated the completion of this work.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT x

CHAPTER

I. Introduction 1

1.1 Motivation 1
1.1.1 The Rise of Cryptographic Ransomware in the Wild 3
1.1.2 The Emergence of Corruption Ransomware in the Digital Realm 5
1.1.3 The Outbreak of Locker Ransomware in Mobile Devices 7
1.2 Research Objectives 9
1.3 Contributions 10
1.4 Dissertation Outline 12

II. Background 14

2.1 Cryptographic Ransomware 14
2.1.1 The Page Cache 15
2.1.2 Measuring Randomness 16
2.2 Corruption Ransomware 19
2.2.1 Corruption Ransomware Features 20
2.3 Locker Ransomware 25
2.3.1 Transformers 26

III. Related Work 29

3.1 Ransomware Detection 29
3.2 Ransomware Recovery 31
3.3 Comparison to Existing Defenses 32

IV. Seamlessly Safeguarding Data Against Cryptographic Ransomware Attacks 35

4.1 Research Objectives 35
4.2 Threat Model 37
4.3 Design and Implementation 38
4.3.1 System Call Interface 39
4.3.2 OS Scheduler 42
4.3.3 The Page Cache Subsystem 47
4.4 Methodology 48
4.5 Evaluation 53
4.5.1 Ransomware Analysis 53
4.5.2 Performance Overhead 57

V. Uncovering Corruption Ransomware: Confronting and Seamless Data Restoration 64

5.1 Research Objectives 64
5.2 Threat Model 65
5.3 The Corruption Ransomware Defense System 67
5.3.1 Ransomware Detection Unit 67
5.3.2 Data Recovery Unit 75
5.4 Experimental Framework 78
5.5 Evaluation 79
5.5.1 Detection and Data Recovery 79
5.5.2 Performance Overhead 80

VI. Sniper: Countering Locker Ransomware Attacks Through Natural Language Processing 90

6.1 Research Objectives 90
6.2 Threat Model 91
6.3 The Sniper Defense System 92
6.3.1 Activity and Service Monitor 93
6.3.2 Text Classification 94
6.3.3 Device Recovery and Pin Restoration 97
6.4 Methodology 98
6.5 Evaluation 102
6.5.1 Model Robustness 102
6.5.2 Workload Characterization 111

6.5.3 Discussion 115

VII. Conclusion and Future Work 118

BIBLIOGRAPHY 120

LIST OF TABLES

Table

3.1	Comparison of our solution with existing ransomware defense systems.	33
4.1	Summary of cryptographic ransomware families and their capabilities.	49
4.2	Summary of benign applications and their respective categories.	51
4.3	Summary of performance workloads.	52
4.4	Configuration parameters of the FIO workloads.	53
4.5	Configuration of the Filebench workloads.	53
4.6	Summary of false positive and negative results.	56
6.1	Summary of hyperparameters used in all models.	99
6.2	Summary of locker ransomware families and their capabilities.	100
6.3	Summary of benign mobile applications and their corresponding categories.	101
6.4	Summary of performance benchmarks.	102
6.5	Summary of quality metrics of NLP models using Non-Adv and DWB, GA and PWWS adversarial test sets.	115
6.6	Summary of detection results.	116

LIST OF FIGURES

Figure

2.1 Entropy of data written to storage as a function of time for WannaCry ransomware, Angry Birds, and Thunderbird. 18
2.2 Data transmission through the network over a TCP connection using different system calls. 21
2.3 Overview of the Data Corruption Process. 22
2.4 Overview of a typical transformer model. 27
4.1 Overview of the system call interface and how different system calls update the tags of their radix trees. 41
4.2 Overview of the OS scheduler design. 43
4.3 A comparison of write entropy measurements across PARSEC, SPLASH-3, and ransomware families over a one second execution period. . . . 54
4.4 Performance impact of our design on compute-bound workloads relative to an unsecured baseline. 58
4.5 Summary of throughput experiments as a function of macro and micro benchmarks from Filebench and FIO. 59
4.6 Summary of IOPS experiments as a function of macro and micro benchmarks from Filebench and FIO. 61
4.7 Summary of latency experiments as a function of macro and micro benchmarks from Filebench and FIO. 63
5.1 Overview of the ransomware defense system. 68
5.2 Overview of the network activity monitoring module. 70
5.3 Overview of the filesystem activity monitoring module. 71
5.4 Overview of the process monitoring module. 73
5.5 Overview of the data restoration module. 77
5.6 Performance overhead of our defense system on compute-bound benchmarks compared to a baseline system. 81
5.7 Summary of micro experiments from FIO benchmarks as a function of throughput, IOPS, and latency. 82
5.8 Summary of micro experiments from Filebench benchmarks as a function of throughput, IOPS, and latency. 84
5.9 Summary of macro experiments from Filebench benchmarks as a function of throughput, IOPS, and latency. 85

5.10 Summary of Netperf experiments as a function of throughput, number of transactions per second, and latency. 89
6.1 Overview of the sniper defense system. 93
6.2 Main components of the natural language processing (NLP) module along with an example of how text from a ransom note is progressively transformed. 95
6.3 Summary of detection rates for non-deep learning models with adversarial and non-adversarial test sets. 103
6.4 Original text example and the corresponding adversarial samples generated using DWB, GA, and PWWS. 104
6.5 Summary of detection rates for deep learning models with adversarial and non-adversarial test sets. 106
6.6 Summary of the runtime for different NLP models. 109
6.7 Summary of the energy for different NLP models. 109
6.8 Breakdown of the runtime overhead for our NLP module phases. . . 111
6.9 Breakdown of the energy overhead for our NLP module phases. . . . 111
6.10 Performance overhead on CPU workloads relative to an unsecure baseline. 112
6.11 Performance overhead on memory workloads relative to an unsecure baseline. 113
6.12 Performance overhead on I/O workloads relative to an unsecured baseline. 114
6.13 Overall performance overhead relative to an unsecured baseline. . . 114

ABSTRACT

In today's interconnected digital landscape, the presence of malware poses a significant threat to individuals, organizations, and society as a whole. Malware, a portmanteau of "malicious software," encompasses a wide range of malicious programs designed to compromise systems, steal sensitive information, disrupt operations, or gain unauthorized access. The proliferation of malware has become a pressing concern, as cybercriminals continually evolve their tactics to exploit vulnerabilities and evade traditional security measures. Malware comes in various forms and can target multiple platforms, including computer systems, mobile devices, and even Internet of Things (IoT) devices. Recently, the digital realm observed the emergence of a new class of malware known as ransomware. Ransomware is a type of malware that involves encrypting, deleting, corrupting the user's data, or locking their device. The malware then extorts the victim to pay a ransom in return for their impacted data or unlocking their device. The impact of such malware can be devastating, resulting in financial losses, operational disruptions, data loss, and irreversible damage. This trend makes it imperative to explore solutions that can seamlessly recover from such malware.

In this work, we examine various ransomware attack variants, including cryptographic, corruption, and locker ransomware. Our investigation focuses on understanding their distinct characteristics, methods of spreading and evading, and the evolving techniques employed by cybercriminals. We propose dynamic solutions that seamlessly safeguard compromised systems from such attacks without the need for a manual recovery process or user intervention. We extensively validate the robustness

of the proposed solutions against thousands of recently released ransomware samples in the wild and show that our solutions reliably recover from all the malicious samples that we tested. Finally, we demonstrate that our proof-of-concept implementation incurs negligible overhead while running a diverse set of realistic workloads commonly used for measuring performance.

CHAPTER I

Introduction

The main objective of this chapter is to provide an introductory overview of the dissertation. The motivation that shows the imperative need for developing defense systems against various ransomware attack variants, such as cryptographic, corruption, and locker ransomware, is explored in Section 1.1. The research objectives of this research are summarized in Section 1.2. The main contributions of this research are detailed in Section 1.3. Lastly, the dissertation organization is outlined in Section 1.4.

1.1 Motivation

In today's interconnected digital world, the existence of malware presents a significant risk to individuals, organizations, and society as a whole. Malware, which stands for "malicious software," encompasses a wide array of harmful programs designed to compromise systems, steal sensitive information, disrupt operations, or gain unauthorized access. The widespread proliferation of malware has become a pressing concern, as cybercriminals continuously refine their strategies to exploit vulnerabilities and evade conventional security measures [1; 2]. Malware takes on various forms and can target multiple platforms, including computer systems [3; 4; 5], mobile devices [6], vehicular systems [7; 8; 9; 10], and even Internet of Things (IoT) devices

[11; 12; 13; 14; 15; 16; 17; 18; 19]. More recently, a new category of malware known as ransomware has emerged within the digital landscape. Ransomware involves encrypting, deleting, corrupting the user's data, or locking their device, followed by a demand for a ransom in exchange for restoring the affected data or unlocking the device.

In recent years, the rise of ransomware attacks has brought a new level of threat and disruption to the world of cybersecurity. Ransomware attacks typically begin with a user unknowingly downloading or executing a malicious file or clicking on a malicious link. Once the ransomware gains access to the victim's system, it begins encrypting, deleting, corrupting files, or locking their device. The victim is then presented with a ransom note that outlines the demands and instructions for making the payment, often in the form of cryptocurrency to maintain anonymity. The motivations behind ransomware attacks primarily revolve around financial gain. Cybercriminals seek to extort money from their victims by encrypting, deleting, corrupting their valuable data or locking their devices and demanding a ransom payment in return for their impacted data or unlocking their devices. The anonymity provided by cryptocurrencies makes it difficult to trace the transactions, making ransomware an attractive option for criminals seeking to monetize their activities. However, the motivations for ransomware attacks can extend beyond financial gain. In some cases, cybercriminals may target specific industries, organizations, or government institutions to cause disruption, damage reputations, or even engage in political or ideological activism. These attacks can be driven by a desire for power, revenge, or the pursuit of certain agendas.

Additionally, ransomware attacks may be used as a cover or distraction for other malicious activities, such as data theft or espionage. By encrypting, deleting, corrupting files, or locking systems, attackers divert attention and resources away from their primary objectives, making it easier for them to infiltrate networks or steal sensitive

information undetected. It's important to note that not all ransomware attacks are carried out by individual hackers. Some attacks may be orchestrated by sophisticated criminal organizations or state-sponsored groups with the aim of gaining financial or political advantages. The impact of a successful ransomware attack can be devastating. For individuals, it may mean the loss of cherished personal files, photographs, or important documents. For businesses, it can lead to the disruption of critical operations, the compromise of sensitive customer data, financial losses, and damage to reputation. In some cases, entire infrastructures, such as hospitals or government agencies, have fallen victim to ransomware attacks, endangering public safety and welfare. The impact of ransomware attacks can be devastating, resulting in financial losses, operational disruptions, data loss, and irreversible damage. This trend makes it imperative to explore solutions that can seamlessly recover from such attacks.

1.1.1 The Rise of Cryptographic Ransomware in the Wild

Our daily dependence on information requires protection. This necessary protection has made data encryption an indispensable technology for preserving the confidentiality of today's digital content. Unfortunately, cybercriminals have discovered ways to re-purpose this technology and deny users access to their data in return for ransom. This trend has sparked an onslaught of ransomware attacks in recent years, resulting in users, businesses, and governments being extorted to pay ransoms in return for restoring their maliciously encrypted data.

According to the U.S. Department of Homeland Security, ransomware represents the fastest growing malware threat to individuals and organizations [20]. Steve Morgan, the founder of Cybersecurity Ventures, painted a grim picture after his public announcement that future ransomware attacks are slated to impact systems every two seconds [21]. To this end, a wide range of business segments incurred significant damages as a result of ransomware, costing the pharmaceutical, shipping services,

and chip manufacturing industries over $850 million, $400 million, and $250 million, respectively [22; 23]. Recently, the energy sector has fallen prey to such attacks after a major U.S. fuel pipeline was taken down, prompting the company to make an immediate ransom payment of $5 million in order to regain access to their encrypted data [24]. Local governments have also fallen victim to similar attacks, adding to ransomware woes, including the City of Baltimore that has spent over $18 million to date in order to recover from ransomware that crippled various municipal operations [25]. Although the cost of ransomware attacks in 2021 has already been estimated to be $20 billion, future damages are projected to reach $265 billion over the next decade [26]. This trend makes it imperative to explore solutions that can seamlessly recover from such attacks.

In response to these challenges, researchers have proposed several defenses to safeguard systems against ransomware attacks that aim to maliciously encrypt user data [27; 28; 29; 30; 31; 32; 33; 34]. A large portion of this research focused on the detection of such malware. For example, Kharraz et al. [27] employed a temporary environment designed to screen user programs. Other solutions [32; 33; 34] proposed monitoring system parameters, such as API calls, registry key operations, and file type changes as features for detecting ransomware activity. However, the response time of such solutions, from data collection to detection, often results in partially encrypted filesystems, leaving victims faced with ransom payment as the only viable option.

The shortcomings of the aforementioned work prompted the research community to investigate solutions that can recover from cryptographic ransomware [30; 31; 28]. Huang et al. [30] explored re-purposing out-of-place writes that are considered intrinsic to flash drives to retain transient information that contain the original plaintext data. Unfortunately, ransomware can overwrite such transient data by duplicating previously encrypted data already present on the drive. Furthermore, the solution is

limited to solid state drives that employ flash technology. Other work [31] proposed the use of function hooking to collect cryptographic keys generated by the OS and saving them to a key escrow. Such keys are retrieved in the event of a ransomware attack to decrypt any impacted files. However, in addition to the potential confidentiality violations this solution introduces, ransomware can bypass this defense by using their own crypto libraries over OS hosted services. In general, recovery solutions such as [30; 31] require manual user intervention where victims can experience long delays when restoring compromised data. Similarly, recovery solutions that employ backups [28] tend to incur non-trivial amounts of performance overhead. Reclaiming compute resources in cloud-based environments where performance overheads are closely monitored to guarantee service level agreements, and user environments where application responsiveness and energy efficiency are treated as first order constraints for consumers are major drawbacks of such systems.

1.1.2 The Emergence of Corruption Ransomware in the Digital Realm

Ransomware's popularity continues to skyrocket due to its profitable business model that enables cybercriminals to receive payment directly from their victims, making it a lucrative activity for ransomware actors. Although ransomware actors previously relied on data encryption tactics as a means to extort payments from victims in return for restoring their maliciously encrypted data, they are now turning to data corruption tactics as they are less time-consuming, less expensive, more efficient, and less likely to be detected by endpoint protection tools [35]. This shift can make victims feel more pressure to pay and lead to higher ransom demands. The trend has emerged as a harmful and well-known type of ransomware that can cause monetary damages to businesses, governments, healthcare institutions, and critical infrastructures.

According to the Department of Homeland Security in the US, ransomware is

the malware threat that is growing the fastest for individuals and organizations [20]. Cybersecurity Ventures founder, Steve Morgan, has painted a gloomy picture, stating that systems are predicted to be impacted by ransomware attacks every two seconds in the future [21]. Recently, a new variation of ransomware has emerged to include data corruption functionalities. The corruption ransomware is a newer version of Exmatter malware, which was previously linked to the BlackMatter ransomware group, and has been updated with data corruption tactics. This tactic was identified in a recent BlackCat ransomware attack [36] and was analyzed by the Cyderes Special Operations team [37] and the Stairwell Threat Research team [38], who show that it indicates a switch from encryption tactics towards corruption tactics. Corruption ransomware attacks have affected tens of thousands of databases across the world [39]. The ransom demands of such attacks can range from $80,000 to $15 million in cryptocurrency, according to a recent advisory from U.S. law enforcement agencies [40] and such attacks can lead to severe financial losses to individuals, businesses, and governments. According to guardian news [41], ransomware payments have increased by nearly double to $1.5 million over the past year. Although it has been estimated that ransomware attacks in 2023 cost $30 billion, future damages are estimated to reach over $265 billion within the next ten years. This trend makes it imperious to discover solutions that can defend against such attacks.

In response to these challenges, researchers have suggested multiple defenses to protect computer systems against encryption-based ransomware attacks [27; 28; 29; 30; 31; 32; 33; 34]. A significant portion of this research has centered on identifying such malware. For instance, Kharraz et al. [27] proposed a virtual environment to monitor user programs' activities. Other solutions [32; 33; 34] relied on monitoring various system parameters that include registry key operations, opcode sequences, API calls, and file type changes to detect ransomware activity. However, these solutions lack the ability to recover from corruption-based ransomware attacks, which

results in partially corrupted filesystems, leaving victims with no choice except to pay the ransom.

To overcome the limitations of the aforementioned solutions, researchers [30; 31; 42; 43; 3; 28] have suggested various recovery solutions in order to recover from encryption-based ransomware attacks. For instance, Huang et al. [30] employed out-of-place writes on solid-state drives in order to restore maliciously encrypted data. Other research [31] suggested using function hooking to gather cryptographic keys generated by the operating system and storing them in a key escrow to use them later for recovering impacted data. Unfortunately, such solutions require manual user intervention, resulting in significant delays in restoring impacted data. Other research [43; 28] proposed backups as a means of recovering maliciously impacted data. However, such solutions tend to result in significant performance overhead and require long periods for recovering impacted data. Unfortunately, the aforementioned defenses lack the ability to recover from corruption-based ransomware attacks.

1.1.3 The Outbreak of Locker Ransomware in Mobile Devices

From virtual reality that swoons users in the metaverse to on the go business carried out by road warriors, smartphones have evolved into the Swiss Army Knife of computing that end users depend on for a wide range of tasks. The year 2022 alone, witnessed more than 255 billion mobile application downloads [44] with 73% of such apps destined to Android devices [45]. Unfortunately, the ubiquity of this platform has garnered significant interest from cybercrime gangs in recent years. For instance, Google officially announced that it removed more than one million risky and untrustworthy applications from its Play Store [46]. This occurred despite all the security checks its applications must undergo before they are made publicly available. Such risks are further amplified by downloads from third party app stores that offer minimal screening for their published apps. An experiment conducted by Allix et

al. [47] demonstrated that up to 75% of application downloads from third party stores included malware. This trend underscores the urgency for solutions that can intrinsically safeguard mobile systems from malicious content.

Unfortunately, the mobile sector has recently fallen prey to a series of aggressive ransomware attacks designed to lock users out of their devices and extort them for payment. From enticing campaigns conceived to ensnare victims to creative obfuscation strategies concocted to evade detection, cybercriminals have demonstrated notable success in compromising mobile devices. According to Kasperky, more than 4.2 million Americans have suffered ransomware attacks on their mobile phones [48]. Another cybersecurity firm confirmed that a slew of smartphone consumers were denied access to their devices after being hoodwinked by CovidLock, an invasive form of locker ransomware that masquerades itself as a legitimate Coronavirus tracker app [49]. Sadly, the woes of this type of malware show no signs of abating. In 2019 alone, an unprecedented 68,000 new mobile ransomware samples were discovered. The sheer number of ransomware variants in recent years has made it challenging for existing solutions to scale their coverage and remain effective. This trend underscores the importance of exploring solutions that can both detect and recover from such attacks.

In response to these challenges, researchers proposed several defenses [27; 50; 51; 33; 52; 43; 28; 53; 54; 55; 56; 57; 58; 59] designed to shield systems from locker ransomware. A large body of this research [27; 50; 51; 33; 52], however, focused on the detection of locker ransomware in traditional computing systems. For example, Kharraz et al. [27] employed a virtual environment to monitor user activity within standard PC systems. Ozsoy et al. [50] suggested the use of hardware performance counters as a way of fingerprinting ransomware behavior. Other solutions [51; 33; 52] leveraged system level features, such as opcode sequences, API calls, and registry key operations for classifying ransomware. Other work examined the detection of this kind of malware on mobile platforms [53; 54; 55; 56; 57; 58; 59]. For instance,

Song et al. [53] explored the aspect of combining different heuristics from the CPU, memory, and I/O subsystems for classifying ransomware. More work [54; 55; 56; 57; 58; 59] investigated the effectiveness of static analysis in exposing ransomware when analyzing resources, such as Dalvik bytecode and XML-based permission files. Unfortunately, these solutions are vulnerable to obfuscation techniques that involve repackaging apps with malicious content directly presented in native machine code [60]. Virtually, all prior work we are of aware lack the ability to recover mobile devices from locker ransomware after infection.

To address the shortcomings of the aforementioned defenses, researchers proposed an assortment of recovery techniques [30; 31; 42; 43; 28]. For example, Huang et al. [30] utilized out-of-place writes in solid-state-drives to restore maliciously encrypted data. Other work [43; 28] suggested the use of backups as a mechanism for recovering maliciously impacted data. All of these techniques, however, are limited to recovering data in systems compromised by cryptographic ransomware. They also require manual intervention by the user and often suffer from long data recovery periods. Virtually, all of the prior work we are of aware lack the ability to recover mobile devices from the effects of locker ransomware.

1.2 Research Objectives

The main objectives of this research are summarized as follows:

- To assess the current landscape of ransomware threats, including its types and characteristics, distribution methods, evasion techniques, and evolving techniques employed by cybercriminals.
- To evaluate existing defense mechanisms, in terms of their effectiveness in detecting, preventing, and recovering from various ransomware attack variants.
- To explore the use of behavioral analysis and artificial intelligence techniques

for defending against different types of ransomware attacks, including cryptographic, corruption, and locker ransomware attacks.

- To develop novel defense systems specifically tailored for combating various ransomware attack variants, including cryptographic, corruption, and locker ransomware attacks.

- To conduct real-world experiments to evaluate the effectiveness and efficiency of proposed defense systems against various types of ransomware attacks, including cryptographic, corruption, and locker ransomware attacks.

- To evaluate the performance overhead of proposed defense systems against a diverse set of realistic workloads commonly used for measuring performance.

- To contribute to the academic and research community by advancing the understanding of ransomware defense, providing insights into emerging trends and future directions for cybersecurity in the face of evolving ransomware threats.

1.3 Contributions

In this work, we propose novel runtime solutions that autonomously defend against various ransomware attack variants, including cryptographic, corruption, and locker ransomware attacks. Unlike prior work that can leave victims with partially impacted filesystems or locked devices, our solutions seamlessly preserve the compromised system without having to undergo an explicit recovery process or manual user intervention. We leverage the use of behavioral analysis and artificial intelligence techniques for defending against different types of ransomware attacks, including cryptographic, corruption, and locker ransomware attacks. We extensively validate the robustness of the proposed solutions against thousands of recently released ransomware samples in the wild and show that our solutions reliably recover from all the malicious samples

that we tested. We also demonstrate that our proof-of-concept implementation incurs negligible overhead while running a diverse set of realistic workloads commonly used for measuring performance.

Overall, this research makes the following contributions:

- Proposes novel dynamic defenses that safeguard the compromised system against various ransomware attack variants, including cryptographic, corruption, and locker ransomware attacks.
- Unlike prior work that can leave victims with partially impacted filesystems or locked devices, our solutions dynamically preserve user system without additional backups or manual intervention.
- Makes the observation that behavioral analysis and artificial intelligence can be harnessed for reliably preserving user system against ransomware attacks.
- Discusses the impact of malicious evasion techniques while demonstrating the robustness of the proposed solutions against such techniques.
- Validates the robustness of the proposed defenses against thousands of recently released ransomware samples in the wild and shows that our solutions reliably recover from all the malicious samples that we tested while tolerating malicious evasion techniques.
- Demonstrates the robustness of the proposed solutions against misclassifying benign programs as ransomware.
- Evaluates the performance overhead of the proposed solutions against a diverse set of realistic workloads commonly used for measuring performance.

1.4 Dissertation Outline

This dissertation is organized as follows: Chapter II presents the background information that is needed for providing a better understanding of the proposed defense systems against various ransomware attack variants, including cryptographic, corruption, and locker ransomware attacks and Chapter III details the related work.

Chapter IV presents a novel runtime solution that seamlessly defends against cryptographic ransomware attacks. This chapter illustrates the design of our defense system that efficiently manages data synchronization between the memory and storage subsystems to prevent maliciously encrypted data from being permanently committed to the underlying storage. It also discusses the robustness of the proposed approach through test it against more than one thousand ransomware samples and shows that the proposed design reliably restores all encrypted files from all ransomware samples we tested including master boot record infection and multi-threaded attacks. Finally, it demonstrates the performance impact of our proof-of-concept implementation through running a mix of compute and I/O bound applications.

Chapter V proposes an end-to-end solution that dynamically defends against corruption ransomware attacks. This chapter illustrates the design of our defense system that combines network activity and filesystem activity monitoring with an efficient data synchronization management system between the memory and storage devices that protects the filesystem from being corrupted. It also discusses the effectiveness of our design against different data corruption tactics and shows that our design effectively recovers all corrupted files. It also demonstrates the robustness of the proposed solution against misclassifying benign applications as ransomware. Lastly, it demonstrates the performance impact of our proof-of-concept implementation has a minimal overhead while executing a wide range of workloads that are typically utilized to measure performance.

Chapter VI presents a novel runtime system that dynamically restores device

access by undoing the effects of locker ransomware attacks. This chapter illustrates the design of our defense system that combines the monitoring of mobile app activity with a natural language processing (NLP) unit that harnesses transformers to detect the appearance of ransom notes. It also discusses the robustness of our solution against more than 5K ransomware samples that span more than 17 families. It also demonstrates the efficiency of the proposed solution to reliably recover from all the malicious samples that we tested, including overlay screen and change pin-code ransomware. It also demonstrates that our experimental proof-of-concept exhibits minimal performance impact while running a mix of mobile benchmark applications. Finally, Chapter VII concludes and discusses the future directions of this research.

CHAPTER II

Background

2.1 Cryptographic Ransomware

Cryptographic ransomware is an invasive form of malware that is designed to purposely encrypt the user's personal information in return for payment. It then coerces the victim to pay a ransom in return for restoring the affected data. Cryptographic ransomware often undergoes four main stages as part of its life cycle: exploitation and infection, delivery and execution, file encryption and backup spoliation, and user notification and clean up. Different methods have been reported for exploiting and infecting users [61]. However, social engineering techniques involving phishing emails, social media, and instant messaging applications are common ways for luring victims into downloading and executing this malware. Once the malware is launched on the system, it communicates with a command and control (C&C) server in order to produce a key pair that will be used for exchanging a private key that will be employed for encrypting the victim's data. Once the key exchange process is completed with the C&C server, the ransomware proceeds to the next stage of scanning the user's filesystem after which it starts the encryption process including the deletion of any backup data (backup spoliation). Upon completion of the encryption process, the payment details and recovery instructions are presented to the victim. The victim is typically given a few days to pay the ransom after which the requested amount

will be increased. Unfortunately, paying the ransom does not necessarily guarantee recovery of the impacted data.

2.1.1 The Page Cache

Advances in process technology fueled by Moore's law have made significant strides in enabling larger memory devices in today's computer systems. However, despite such advances, designers still need to balance between the use of low latency technology, storage capacity, and cost as part of producing affordable, yet high performing computer systems. This goal often necessitates designers to construct a hierarchy of memory devices from an eclectic mix of technologies that possess different speed and capacity characteristics. These devices are often interlinked with one another through caches.

Caches are designed to retain copies of fetched data obtained from lower levels of the memory hierarchy under the premise that recently used data will likely be re-used in the near future. This allows upstream levels of the memory hierarchy to minimize access to downstream devices that are considered slow. To this end, virtually all modern operating systems rely on caching disk data into main memory as a way of improving performance. This is primarily driven by the fact that accessing main memory is several orders of magnitude faster than accessing a disk drive (nanoseconds vs. milliseconds) [62].

Page caches consist of physical blocks of data that are fetched from non-volatile storage media (backing store) such as magnetic disks and solid state drives. Whenever a process issues a `read()` system call, the kernel searches its page cache for the data, if found, the present data is used to service the calling process. However, in the event that the data is not found within the cache, an I/O transaction is issued to the backing store to obtain the requested blocks. Once fetched, these blocks are populated as pages within the cache in order to promote isolation between any active

processes through the use of virtual memory. Populating the cache with this data obviates the need for making future I/O requests to the backing store for subsequent transactions to the same data.

Write Caching. In addition to servicing read requests, page caches must efficiently handle write transactions without invalidating present pages every time data is modified. In order to address this issue, page caches often use a write-back policy for synchronizing data to the backing store. In other words, the backing store is not immediately updated. Instead, modified pages are marked dirty and synchronized periodically to make the corresponding data in the backing store up to date with the version present in the page cache. Operating systems often dedicate one or more threads for systematically scanning dirty pages and synchronizing them with the backing store.

Cache Eviction. Main memory is a valuable resource that running applications often contend over. As a result, the page cache must dynamically adapt its footprint to make room for incoming data when new entries are no longer available or when the available memory of the overall system is low. In order to achieve this, page caches implement a cache eviction policy that determines which pages must be removed from the cache. As a first step, OS's target clean pages for eviction. However, if more memory needs to be freed, the operating system creates more clean pages by writing dirty pages back to the disk then evicting them from the cache. The OS chooses pages that are the least likely to be used in the future. For this OS's often use a *least recently used* approach where pages are evicted according to their timestamp.

2.1.2 Measuring Randomness

Cryptographically strong algorithms aim to promote two fundamental properties: confusion and diffusion. Confusion relates to an algorithm's ability to obscure the relationship between a cryptographic key κ and the ciphertext $\mathcal{C}$ it produces [63].

Diffusion, on the other hand, focuses on obscuring the statistical properties of the plaintext message $\mathcal{M}$, making the produced ciphertext $\mathcal{C}$ appear random relative to its original data $\mathcal{M}$. The aforementioned property can also be harnessed to infer the presence of ciphertext $\mathcal{C}$ within a stream of data. In other words, the amount of randomness present in a given bytestream can serve as an indicator for distinguishing plaintext from encrypted data.

Various methods have been proposed for assessing randomness [64]. A commonly used algorithm for quantifying the amount of randomness present in a block of data is the information entropy test. A metric originally introduced by the father of information theory, Claude Shannon, entropy is concerned with measuring the degree of uncertainty in a set of bytes. As such, digital content that undergoes encryption (ciphertext) tends to exhibit consistently high levels of entropy. Our study makes use of this metric for detecting encryption activity initiated by ransomware which is described in equation (2.1). In this equation, $p(b_i)$ denotes the probability of byte value b_i occurring in a given block of data $\mathcal{M}$ consisting of n bytes. This metric yields values $H(B) \in [0, 8]$, with 8 corresponding to a stream of data that has a perfectly even distribution of byte values. Since bytes within a ciphertext should have a uniform probability of occurring, encrypted data tends to approach the upper bound of this range.

$$H(B) = -\sum_{i=0}^{n-1} p(b_i) \log_2 p(b_i) \tag{2.1}$$

In order to evaluate the suitability of using entropy as a metric for classifying ransomware, we conducted a series of experiments that measured the entropy of data written to the disk while executing different programs. To this end, we first launched WannaCry, a ransomware program responsible for $4 billion in damages to a wide range of industries after maliciously encrypting data on more than 230K computers [65]. We executed a sample of the aforementioned ransomware and monitored its disk

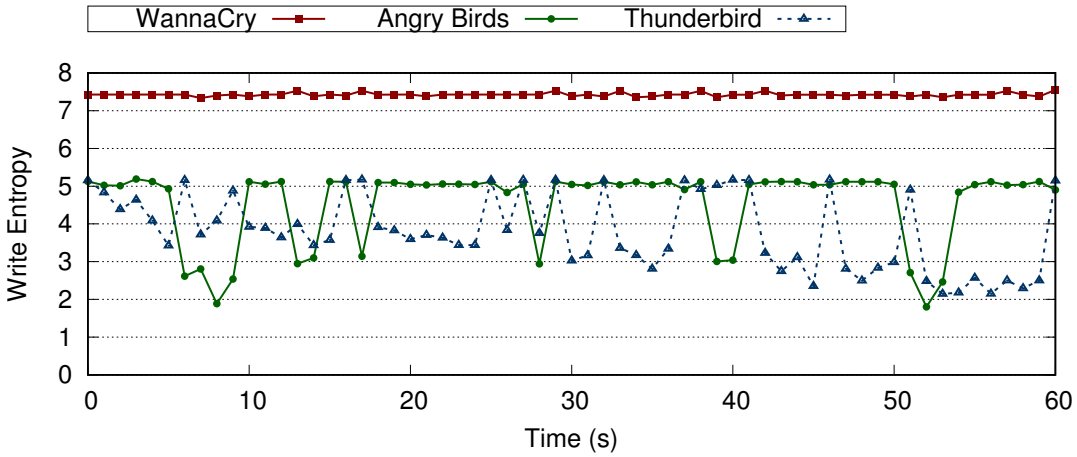


Figure 2.1: Entropy of data written to storage as a function of time for WannaCry ransomware, Angry Birds, and Thunderbird.

activity over a period of one minute. We also examined the disk activity associated with two other benign applications over the same period. We ran Angry Birds, a popular gaming application, and Thunderbird, another popular email client. The results of this experiment are illustrated in Figure 2.1.

We observe that on an x86 platform equipped with four CPU cores, 8 GB of memory, and 1 TB of storage, an average entropy of 7.4 was recorded while monitoring the write transactions WannaCry issued to the disk. We find that this rate is sustained over the entire one minute execution window. The lowest entropy value that we recorded during this one minute period was 7.3. We observe a different trend, on the other hand, while running the gaming application, Angry Birds. This application averaged an entropy value of 4.6. Furthermore, the entropy never exceeded 5.2 throughout the execution phase while actively playing games through this program. We observe a similar trend while interacting with the email client, Thunderbird. This application exhibited an even lower average of 3.8 throughout its execution. Similar to Angry Birds, the entropy value never peaked beyond 5.2. This data illustrates the potential of this metric in distinguishing between ransomware and benign applications.

2.2 Corruption Ransomware

Corruption ransomware is an invasive form of malware that is designed to purposely corrupt the data of its victims and demand a ransom in exchange for restoring the affected data. Typically, the malware coerces the victim into making the payment using cryptocurrency-based services since they provide a high level of anonymity for financial transactions. Regrettably, there is no guarantee that paying the ransom will result in the successful restoration of the affected data. Unlike cryptographic ransomware, corruption ransomware is faster, cheaper, and less likely to be detected by endpoint protection tools. The development of corruption ransomware code is also notably easier as there is no need to worry about reversing the damage after the victims pay the ransom. Ransomware corruption tactics further hinder the progress of security researchers in creating decryptors that exploit weaknesses in cryptographic ransomware tactics as seen in the cases of the Lorenz [66] and MafiaWare666 [67] ransomware variants. In summary, data corruption tactics enable ransomware actors to save time and effort while increasing the likelihood of a successful ransom payment.

Corruption ransomware often undergoes five main stages as part of its life cycle: exploitation and infection, delivery and execution, file transferring and queuing, file corruption and backup spoliation, and user notification and extortion. Different methods have been reported for exploiting and infecting users [61]. However, social engineering techniques involving phishing emails, social media, and instant messaging applications are common ways for luring victims into downloading and executing this malware. Once the malware is launched on the system, the malware systematically scans the victim's filesystem, creating a list of files that correspond to specific predefined extensions. These files are then transferred to a command and control (C&C) server controlled by the attacker. Once the files are uploaded to the C&C server, the successfully transferred files are placed in a queue for the corruption process [68]. The ransomware proceeds to the next stage of transferring and queuing the victim's

files after which it starts the corruption process including the deletion of any backup data (backup spoliation). Upon completion of the corruption process, the payment details and recovery instructions are presented to the victim. The victim is typically given a few days to pay the ransom after which the requested amount will be increased. Unfortunately, paying the ransom does not necessarily guarantee recovery of the impacted data.

2.2.1 Corruption Ransomware Features

Data Transmission. After the corruption ransomware is delivered and launched on the victim's system, the ransomware starts with scanning the victim's filesystem in order to create a list of files that correspond to specific predefined extensions (e.g., .txt, .pdf, .docx, .pptx, .xlsx,..., etc.). The created list of files is then transferred through the network over a TCP connection to ensure a reliable transmission to a command and control (C&C) server controlled by the attacker. The act of transferring the victim's files to the C&C server controlled by the attacker is a common tactic employed by corruption ransomware. This tactic is typically accomplished by transmitting the victim's files across the network via a TCP connection, employing one of the data transmission methods depicted in Figure 2.2.

The first method showcases the process of transmitting the victim's files through the network over a TCP connection via data streaming. This is achieved by utilizing `read()+write()/send()` system calls as depicted in the left side of Figure 2.2. As observed in the left side of Figure 2.2, the file being transmitted from the disk to the C&C server needs to pass through the buffer cache, which is located in the kernel space. Subsequently, the file's data stream is copied into the memory area of the user-space buffer using `read()` system call before being written or sent back to a newly established network TCP socket. Afterward, the file's data stream is copied again from the memory area of the user-space buffer to the kernel socket send buffer using

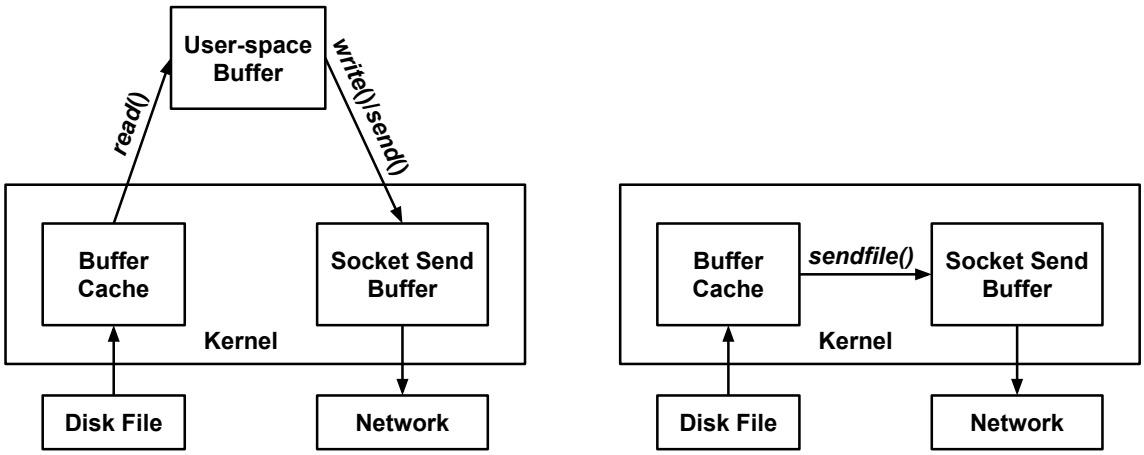


Figure 2.2: Data transmission through the network over a TCP connection using different system calls.

`write()/send()` system calls before being ultimately transmitted over the network.

On the other hand, the second method illustrates the transmitting process of the victim's files across the network via a TCP connection by transmitting the entire files. This is accomplished by employing the `sendfile()` system call as shown in the right side of Figure 2.2. Unlike the first method, which involves using `read()+write()/send()` system calls and relies on an intermediate user-space buffer for data transmission, this method employs the `sendfile()` system call that is designed to facilitate data transfer solely within the kernel space. Consequently, this method has the capability to send the entire file's data in a single operation. This implies that the file being transmitted from the disk to the C&C server is directly transferred from the kernel buffer cache to the socket send buffer using the sendfile() system call before being ultimately transmitted over the network.

Data Corruption Tactics. After the victim's files are effectively transmitted and uploaded to the C&C server controlled by the attacker, the successfully transmitted files are subsequently queued for the corruption process. The ransomware then starts with deleting any existing backup data (backup spoliation) in order to avoid restoring from the backup and make recovery more difficult. Following that, the ransomware

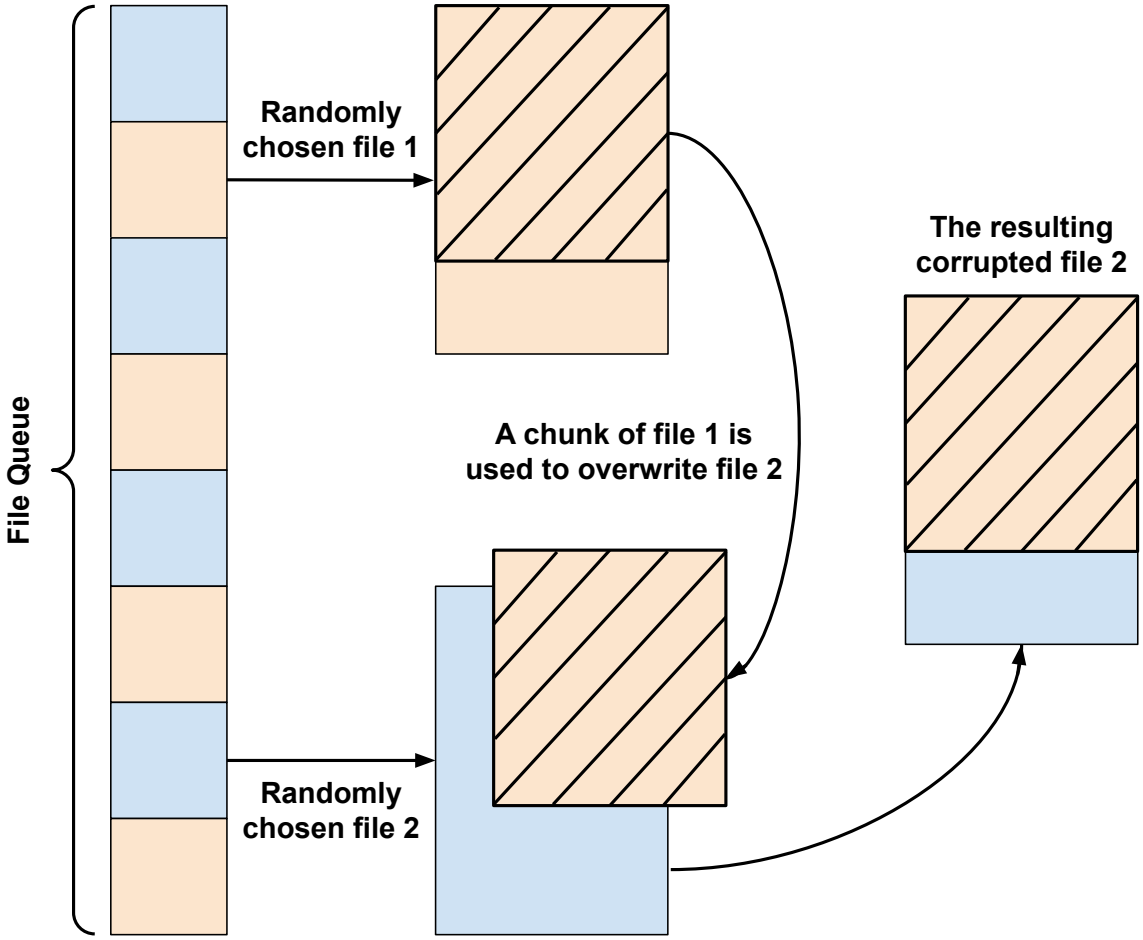


Figure 2.3: Overview of the Data Corruption Process.

proceeds to start with the actual damage process, which involves the data corruption of the queued files on the compromised system. The process of data corruption begins by randomly choosing two files from the corruption queue, where the first file serves as the source for overwriting and corrupting the contents of the second file. Once the two files are chosen, a data chunk of a random size which can be as small as a single byte is chosen from the beginning of the first file and read into a buffer. This chosen data chunk is then written into the beginning of the second file, resulting in its effectively overwriting and corrupting [68]. An overview of the data corruption process is shown in Figure 2.3.

In addition, corruption ransomware employs a variety of tactics to corrupt the

victim's files on the compromised system. These tactics are employed in order to expedite the data corruption process and avoid detection. In particular, we extract the following six tactics that are employed by corruption ransomware for data corruption:

- **Data Swapping.** This tactic relies on swapping the data between two randomly chosen files from the corruption queue. The process begins with the random selection of two files from the corruption queue. Subsequently, two data chunks are randomly chosen from the two files and stored in a buffer. The data chunk selected from the first file is then written into the beginning of the second file, while the data chunk selected from the second file is written into the beginning of the first file. This leads to the effective overwriting and corruption of the two files [68]. This tactic allows ransomware to speed up the data corruption process compared to data encryption and avoid detection by endpoint protection tools.

- **Partial Encryption.** This strategy depends on partially encrypting the data of a randomly selected file from the corruption queue. The process starts with randomly selecting a file from the corruption queue. Then, a specific portion of the data from the file is selected for encryption. Consequently, the selected file becomes effectively corrupted [69]. This strategy allows ransomware to expedite data corruption compared to full encryption. The time saved is particularly noticeable when dealing with large files, as the encryption process per file is reduced to a matter of minutes. This strategy also avoids ransomware from being detected by endpoint protection tools that utilize techniques such as Chi-squared or data entropy measurements.

- **Data Distortion.** This technique relies on overwriting the data of a randomly chosen file from the corruption queue with null bytes, zeroes, or garbage values. The process begins by randomly selecting a file from the corruption queue, and

then a random portion of the file's data is chosen to be overwritten with null bytes, zeroes, or garbage values. As a result, the chosen file becomes effectively corrupted [68]. This technique allows ransomware to accelerate data corruption in comparison to data encryption and evade detection by endpoint protection tools that employ methods like Chi-squared or data entropy measurements.

- **Data Deletion.** This strategy relies on the deletion of randomly selected files from the corruption queue. This is often accomplished by generating corrupted copies of selected files and subsequently deleting the corresponding original files, or by deleting the original files after they have been successfully transmitted and uploaded to the C&C server controlled by the attacker [70]. This strategy allows ransomware to expedite the corruption process and evade detection.

- **Multi-threaded Approach.** This approach relies on leveraging the capabilities of multiple core processing units present in contemporary CPUs, coupled with hyper-threading features. Ransomware leverages this technology to distribute the corruption activities of ransomware across multiple cores. This approach allows ransomware to speed up the corruption process and evade detection to surpass any preventive measures that victims or system administrators may initiate in response [71].

- **Slowing Tactic.** In contrast to the multi-threaded approach, which spreads the corruption activities of ransomware across multiple cores to accelerate execution and evade detection, this tactic depends on intentionally slowing down the corruption process. It accomplishes this by writing data at low I/O rates which can be as low as a single byte per second in order in order to avoid detection. This technique allows ransomware to evade detection by detectors that employ methods like I/O rate measurements [72].

2.3 Locker Ransomware

Locker ransomware is a form of malware that maliciously appropriates a victim's device while denying access to its legitimate user. On mobile systems, this is accomplished by changing the device's pin code upon gaining root privilege. Another common approach is to permanently overlay a new window that prevents access to the system resources, as well as user installed apps. The malware then extorts the victim to pay a ransom in return for restoring access to the infected device. Payment is often accomplished through cryptocurrency based services since they offer end-to-end anonymity of financial transactions. Unfortunately, paying the ransom does not always guarantee restoration of the infected device.

Locker ransomware generally undergoes five primary stages as part of its lifecycle. These stages include: exploitation and infection, delivery and execution, information exchange, destruction, and extortion. A wide range of methods have been reported for the initial stage of exploitation and infection [73]. However, a combination of social engineering techniques that leverage phishing through social media, instant messaging, and seemingly benign apps, are common ways for luring victims into launching ransomware onto their devices [47]. Once such malware is deployed on a given device, it swindles the victim into granting it administrator privileges. At this point, the malware proceeds to the next stage by establishing a communication channel with a command and control (C&C) server. The channel is mostly used to share details about the infected device and retrieve a set of commands to be executed on the target. Once the necessary information has been exchanged between the C&C server and the infected device, the malware proceeds to the destruction stage that results in the denial of access to the device. The process is concluded by presenting a ransom note to the victim that describes the payment details and recovery instructions.

2.3.1 Transformers

Transformers have demonstrated significant promise in solving NLP tasks [74; 75; 76]. A core component of this architecture relates to its encoder-decoder structure that consists of multi-head attention and fully connected feed-forward network layers. An overview of a standard transformer model is shown in Figure 2.4 [74].

Encoder and Decoder Stacks. The encoder serves the purpose of mapping sequences of symbols $(x_1, ..., x_n)$ to a sequence of continuous representations $z = (z_1, ..., z_n)$. It consists of N identical blocks where each block contains two layers: multi-head attention and position-wise fully connected feed-forward network layers. Each of these layers is surrounded by residual connections and followed by layer normalization in order to speed up training. Input is presented in the form of positional encoded word embeddings in order to keep track of the order of the input sequence. The decoder, on the other hand, assumes the task of generating the output sequence of symbols $(y_1, ..., y_n)$, from a sequence of representations z. Unlike the encoder, the decoder includes a third layer that performs multi-head attention on the encoder's output.

Attention. Attention is the process of mapping a query and key-value pairs to an output, where the query, keys, values, and output are all vectors. The attention within a given transformer that involves a query, key, and value matrices Q, K, and V, is computed using the scaled dot-product illustrated in equation (2.2). The $\sqrt{d_k}$ term is used for scaling where d_k is the dimension of the input K. This mechanism of computing the attention is repeated several times in parallel for different linear projections of the input. The output is then concatenated and multiplied by an additional weight matrix resulting in what is called multi-head attention. The output of the multi-head attention layer is then used as input into a position-wise fully connected feed-forward network layer.

$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.2}$$

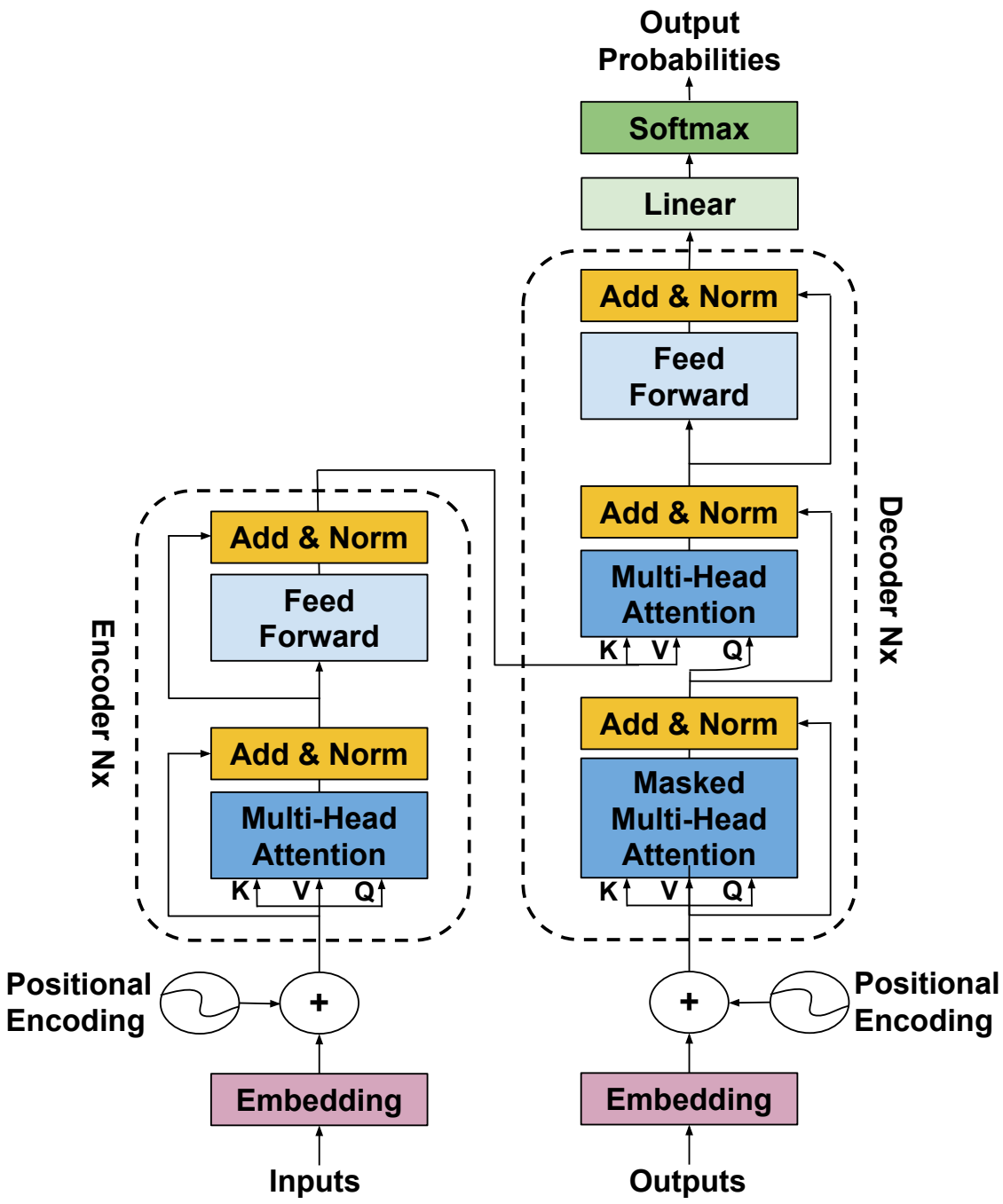


Figure 2.4: Overview of a typical transformer model.

Position-wise Feed-Forward Network. The encoder and decoder of a given transformer contain a fully connected feed-forward network. This step consists of two linear transformations with a ReLU activation in between. The position-wise feed-forward network is expressed using equation (2.3).

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2.3}$$

Embeddings and Softmax. Input and output tokens are converted to dimensional vectors using learned embeddings. The final decoder output runs through a learned linear transformation followed by a softmax layer to convert the outputs to subsequent token probabilities.

After the introduction of transformers [74], [75] proposed the use of Bidirectional Encoder Representations from Transformers (BERT) as an enhancement for solving NLP tasks. This architecture consists of a bidirectional encoder that learns information from both the left and right sides of a word's context during the training phase. In addition to enhancements made by BERT, a more recent architecture was introduced, namely, XLNet [76]. Unlike BERT, XLNet leverages a generalized autoregressive pre-training method that enables learning bidirectional contexts more efficiently. Our work examines both of these architectures and evaluates their suitability for detecting ransom notes.

CHAPTER III

Related Work

3.1 Ransomware Detection

PC Ransomware Detection. A large body of research [32; 27; 29; 77; 34; 78; 33; 79; 80] explored various detection techniques that are aimed at defending against ransomware attacks. For example, Kharraz et al. [27] proposed a dynamic analysis framework that used an artificial environment for identifying ransomware activity. Other work explored the use of decoy files as a feature for detecting malicious accesses to the filesystem [81; 28; 82]. More recent work [83; 84] expanded on the aforementioned approach by monitoring a range of content and behavior-based features for fingerprinting ransomware activity. Other techniques [33; 34; 79; 80] proposed monitoring system parameters such as API calls and file type changes as features for detecting malicious activities. Other solutions [85; 52], on the other hand, suggested the use of opcode sequences as features for performing the detection. Other solutions [86; 50], suggested the use of hardware performance counters to detect malicious activities. However, it has been demonstrated that such techniques are unreliable for predicting malicious activity [87]. Other work by Scaife et al. [32] focused on the use of an early-warning detection system that was designed to notify users whenever a process appeared to tamper with large amounts of data. Other solutions [33; 34] proposed monitoring system parameters, such as API calls, registry key operations,

and file type changes as features for detecting suspicious activity. On the other hand, Moore [82] et al. explored the use of honeypots. The authors achieved this by developing a system that tracked changes to a honeypot folder that was designed to lure ransomware into disclosing its runtime behavior. Other work by Continella et al. [28] proposed a system known as ShieldFS. The system leveraged an add-on driver to enable monitoring low-level filesystem activity. The activity was then logged and fed into a model that was tuned to classify ransomware activity.

In addition, multiple bodies of work examined the use of machine learning as a defense against ransomware attacks [88; 51; 79; 80; 52; 85; 89; 90; 91; 92; 93]. For example, EldeRan [51], a solution proposed by Sgandurra et al., leveraged machine learning to infer ransomware activity based on early actions taken during the installation process. Chen et al. [88] proposed a data mining technique for detecting ransomware in a virtual environment based on API calls. On the other hand, work by Takeuchi et al. [79] and Vinayakumar et al. [80] proposed a detection scheme that used API calls as features. More recent work [52; 85], on the other hand, considered the use of opcode sequences, instead, as features for performing the detection. Although high detection rates have been reported for many of these solutions, the response time of the aforementioned solutions, from data collection to detection, often results in partially impacted filesystems or locked devices. Therefore, leaving victims faced with ransom payment as the only viable option. Unlike prior work, however, our solution goes beyond ransomware detection. It instead focuses on undoing the effects of ransomware attacks after a system has been infected.

Mobile Ransomware Detection. A large body of research [29; 55; 56; 58; 94; 59; 95; 57; 96; 97; 53; 54; 98; 99] explored various detection techniques against Android ransomware. For example, RansomProber [29] relied on UI analysis on mobile devices to distinguish between benign and malicious behavior. Other techniques [55; 95; 57; 96; 97; 56; 58] used different forms of static analysis including API calls,

text embedded within APK files, and Dalvik bytecode present in DEX files for performing detection. Unfortunately, such solutions can be thwarted if malicious content is encrypted or their languages are changed [100]. More work [94; 59] considered permission sequences for fingerprinting malicious content. In general, the aforementioned methods rely on analyzing high-level bytecode or permissions files, which are ineffective against obfuscation techniques that involve repackaging apps with malicious content directly in the native form [60]. Other work by Lachtar et al. [101; 92; 93] proposed detecting ransomware at the native instruction level. However, such work lacks recovery capability.

In addition, Song et al. [53] proposed a dynamic solution that examines system resources associated with a given process including CPU, memory, I/O activity. However, relying on usage thresholds have been shown to unreliable in predicting malicious activity [87]. Other work [99] proposed Dna-Droid, a hybrid approach in which apps are initially assessed statically by examining text, images, API calls, and permissions present within an APK file. API call sequences are then monitored at runtime in the event that the static analysis phase marks a given app as suspicious. However, static analysis solutions are vulnerable to obfuscation different techniques [100; 60] that can render the dynamic layer ineffective. Overall, all prior work we are aware of are incapable of recovering systems from ransomware attacks after the system has been infected, leaving victims with ransom payment as the only viable option. Unlike prior work, we propose an end-to-end solution that seamlessly safeguards compromised systems from ransomware attacks without the need for a manual recovery process or user intervention.

3.2 Ransomware Recovery

To address the shortcomings of prior work, various researchers proposed different recovery techniques as a solution against ransomware attacks [31; 102; 42; 103; 30;

104; 105; 106; 43; 28]. For instance, Kolodenker et al. [31] proposed a framework that relied on a key vault designed to retain all cryptographic keys produced on the system. Such keys are then retrieved to decrypt any affected files in the event that a system becomes infected. Other work by Subedi et al. [42] proposed isolating backup data away from the device's standard volume and making it inaccessible to ransomware in order to mitigate backup spoliation attacks. Finally [103; 30; 104; 105; 106], focused on harnessing the intrinsic properties of flash drives. For example, work by Huang et al. [30] proposed a framework that leveraged out-of-place writes that are inherent in solid-state-drives to recover encrypted data. However, such recovery solutions require manual user intervention where victims can experience long delays when restoring compromised system. Similarly, recovery solutions that employ backups [28] tend to incur non-trivial amounts of performance overhead. Unlike prior work that can leave victims with costly downtimes that stem from long data recovery periods, our solution seamlessly preserves compromised system without having to undergo an explicit recovery process. Furthermore, our work does not require retaining any confidential data in order to undo the effects inflicted by ransomware attacks.

3.3 Comparison to Existing Defenses

We compared our solutions against eight notable defenses that represent the state-of-the-art in ransomware detection and recovery. The differences in terms of dataset size, detection metrics and capability, features, real-time recovery, and performance overhead are outlined in Table 3.1. In general, our evaluation consisted of executing a larger number of ransomware samples relative to other work [27; 83; 32; 51; 84; 28; 31]. With the exception of [30], our dataset size for ransomware is $2\times$ – $12\times$ larger than what was considered in the other studies. Although FlashGuard [30] used a slightly larger dataset size of 1,477 samples, the study only covered a total of 13 ransomware families. Unlike Flashguard [30], our work considers samples across 18 recently re-

Solution	**Dataset**	**FP**	**FN**	**FA**	**NA**	**PA**	**MBR**	**File Recovery**	**Realtime Recovery**	**I/O**
UNVEIL	319	0.0%	0.0%	✓	✗	✓	✗	✗	✗	-
Redemption	677	0.8%	0.0%	✓	✗	✓	✗	✗	✗	5.6%
CryptoDrop	492	3%	0.0%	✓	✗	✓	✗	✗	✗	-
ElderRan	582	1.6%	3.7%	✗	✗	✓	✗	✗	✗	-
RWGuard	261	0.1%	0.0%	✓	✗	✓	✗	✓partial	✗	1.9%
ShieldFS	305	0.0%	2%	✗	✗	✓	✗	✓	✗	380%
PayBreak	107	-	-	✗	✗	✗	✗	✓partial	✗	150%
FlashGuard	1,477	-	-	✗	✗	✗	✗	✓	✗	$< 6\%$
Our Work	**1,324**	**0.0%**	**0.0%**	✓	✓	✓	✓	✓	✓	**$< 1\%$**

Table 3.1: Comparison of our solution with existing ransomware defense systems.

leased ransomware families. For example, our evaluation included testing NoMoreRansom, a variant of the Rapid family that was released in 2020. Furthermore, in terms of detection rates, our work yielded no false positives (FP) or false negatives (FN) throughout the evaluation. The only work that had similar results is UNVEIL [27]. However, unlike UNVEIL [27], our work considers over 4× the amount of ransomware samples and covers more than 2× the number of families. The remaining studies [83; 32; 51; 84; 28] exhibited a combination of false positives and negatives. These results demonstrate the practicality of our solution and its ability to distinguish between benign and malicious workloads. We attribute this to the diverse set of feature classes our study relies on for detection. For instance, our work uses features that include monitoring at the filesystem activity (FA), process activity (PA), and network activity (NA). Although other work considered using a combination of filesystem and process activity monitoring for detecting ransomware, none of the studies listed in Table 3.1 considered network activity as a detection feature.

Another aspect we examined in our comparison relates to the ability to detect master boot record (MBR) infecting ransomware. In addition to detecting standard and multi-threaded ransomware, our solution was able to detect MBR-based ransomware. However, unlike our work, none of the other defenses listed in Table 3.1

demonstrated the ability to protect against such malware. In addition to MBR infecting ransomware, we compared our solution to other defenses in terms of recovery capability. Our defense was able to seamlessly restore all impacted files in real-time without requiring the user to undergo an explicit recovery process. Our solution was able to achieve this across more than one thousand ransomware samples obtained from 18 families. This was accomplished while demonstrating robustness against evasion techniques that include MBR infecting ransomware, multi-threaded ransomware, as well as smart ransomware. Although solutions, such as RWGuard [84] and PayBreak [31] offer system recovery, this recovery is dependant on ransomware using the Microsoft CryptoAPI. As such, the aforementioned solutions cannot recover system that are encrypted by ransomware that employ custom-written libraries. Solutions, such as ShieldFS [28] and FlashGuard [30], on the other hand, offer full recovery of impacted system. However, the aforementioned defense incur a non-trivial amount of performance overhead. They also require the system to undergo an explicit recovery process. This manual user intervention can result in victims experiencing long delays before any impacted system are restored. Finally, the performance overhead introduced by our solution to the I/O subsystem is negligible compared to other defenses. Our solution incurs less than 1% overhead to I/O transactions. This is significantly less than what was reported for the studies listed in Table 3.1. In addition to I/O overhead, our solution characterizes the computation overhead associated with our defense. This is important since any code that analyzes ongoing system activity in order to make timely decisions, requires appropriating CPU cycles from compute bound applications. Similar to I/O transactions, our computation overhead is relatively small (less than 2%). Other studies, on the other hand, do not explicitly evaluate such overhead. Therefore, we do not discuss this in Table 3.1. However, we note that multiple studies including [84; 51; 28] employ machine learning algorithms as part of their detection. Such algorithms have a non-trivial amount of computation.

CHAPTER IV

Seamlessly Safeguarding Data Against Cryptographic Ransomware Attacks

4.1 Research Objectives

In this chapter, we propose a novel runtime solution that autonomously defends against cryptographic ransomware. Unlike prior work that can leave victims with partially encrypted filesystems or costly downtimes that stem from long data recovery periods, our solution seamlessly preserves compromised data without having to undergo an explicit recovery process. A key observation made by this work is that maliciously encrypted data is initially buffered in the operating system's page cache before it is flushed to the underlying storage device. Based on this observation, we develop a solution that efficiently manages data synchronization between the memory and storage subsystems to prevent maliciously encrypted data from being permanently committed to the underlying storage. We extensively validate the robustness of this approach against more than one thousand recently released samples that span 18 ransomware families. We show that our design reliably restores all encrypted files initiated by the samples we tested. Furthermore, our solution is resilient to ransomware that employ techniques including master boot record infection [107] and multi-threaded attacks [71]. We demonstrate that our proof-of-concept

implementation incurs negligible overhead while running a diverse set of realistic workloads commonly used for measuring performance. Evaluation using a range of benchmarks that include: PARSEC [108], SPLASH-3 [109], Flexible I/O [110], and Filebench [111], show an average performance degradation that is less than 2% across compute-bound workloads and less than 1% across I/O bound workloads.

Overall, this chapter makes the following contributions:

- Presents a novel defense that safeguards data against cryptographic ransomware.
- Unlike prior work that leaves filesystems partially encrypted or requires delays for recovering data, our solution dynamically preserves user data without additional backups or manual intervention.
- Makes the observation that page caches that are commonly employed by operating systems to buffer I/O data can be harnessed for reliably preserving storage devices against ransomware attacks.
- Proposes an efficient end-to-end runtime system that incurs minimal overhead across a diverse set of realistic workloads in the form of micro and macro benchmarks.
- Extensively tests the robustness of our solution against more than one thousand recently released ransomware samples and demonstrates that our work reliably restores all encrypted files while tolerating malicious techniques such as master boot record infection and multi-threaded attacks.

The rest of this chapter is organized as follows: Section 4.2 discusses the threat model. Section 4.3 describes the design of the proposed system. Section 4.4 presents the methodology and experimental framework used in this work and Section 4.5 discusses the results of our evaluation.

4.2 Threat Model

Our design aims to safeguard computer systems against cryptographic ransomware that is crafted to maliciously encrypt data present on a victim's filesystem. To this end, we consider malware empowered with capabilities beyond traditional ransomware applications. Such capabilities include the ability to infect the master boot record (MBR), as well as, perform concurrent execution through multi-threading. To this end, we assume an attacker can deploy ransomware that belongs to any of the following categories:

- **Standard Ransomware.** Attackers launch ransomware on a victim's platform, typically by leveraging social engineering techniques such as phishing. The malware is assumed to have basic ransomware capabilities including the ability to scan a filesystem and encrypt its data through a cryptographically strong algorithm.

- **MBR Infecting Ransomware.** Similar to standard cryptographic ransomware that can scan and encrypt data on a filesystem, this form of malware also has the ability to infect the master boot record, prevent the OS from booting, and redirect the boot process to a malicious bootloader that locks the victim out of the system [112].

- **Multi-threaded Ransomware.** Such ransomware harnesses the power of multi-core processing units commonly found in modern CPUs and other features such as hyper-threading (HT). Attackers can leverage this technology for distributing ransomware activity across multiple cores to evade detection, as well as, speed up execution in order to outpace any preventative response that a victim or system administrator may initiate [113; 71].

In addition to the above, we assume ransomware can encrypt user data by directly

overwriting existing files using in-place writes or by creating encrypted copies of existing files through out-of-place writes. We also assume that maliciously encrypted copies of the user data can be followed by the deletion of the associated original files. Our design also assumes that ransomware can perform data encryption using any cryptographic algorithm that is either directly embedded within the malware or available through crypto services that are provided by the OS. For instance, most ransomware employ the Advanced Encryption Standard (AES) algorithm for encrypting user data. However, we assume an attacker can employ other encryption algorithms beyond AES including Rivest-Shamir Adleman (RSA), Elliptic-curve cryptography (ECC), and Rivest Cipher 4 (RC4). Furthermore, our solution is designed to restore encrypted files irrespective of the underlying storage technology. As such, our work is able to safeguard data present on a variety of storage media including solid-state drives and magnetic spin disks. Finally, we make the assumption that the OS is trusted and free from any privilege escalations. Otherwise, we argue that any in-host defense would be defeated including anti-malware solutions. As such, we assume that ransomware is executed in user-mode.

4.3 Design and Implementation

We present a new runtime system that dynamically defends against the effects of cryptographic ransomware. A key approach to our design is that maliciously encrypted data is initially buffered in the operating system's page cache before it is flushed to the underlying storage device. Starting from this observation, we propose an end-to-end solution that efficiently manages data synchronization between the memory and storage subsystems to prevent maliciously encrypted data from being permanently committed to the underlying hard drive. To support this approach, we implement our solution within the operating system through modifications to the system call interface, scheduler, and page cache.

4.3.1 System Call Interface

Our solution is designed to track I/O transactions that could result in the malicious modification of files present on the system and prevent them from reaching the backing store. To this end, we augment the system call interface to monitor `socket()`, `write()`, and `delete()` operations issued from user space.

Network Socket Requests. All user space processes are initially treated as benign until they are determined otherwise, based on their system call activity. An important characteristic of cryptographic ransomware is that it communicates with a command and control (C&C) server in order to exchange the key it will consume for encrypting the victim's data [114]. This encryption key is used later by the C&C server to demand a ransom. As such, our design tracks processes that attempt to communicate over the network. Whenever a process issues a request to the kernel to create a network socket via the `socket_create()` system call, our solution updates a `tgid_ransom_t` data structure that we associate with the requesting process. This structure is shown in Listing IV.1. More specifically, the `tgid_ransom_t` of the requesting process is updated to reflect that a socket is being created using the `socket_created` field. Monitoring the `socket_create()` system call, allows us to flag ransomware that attempts to communicate with a C&C server over TCP and UDP connections.

Write Requests. Whenever a process issues a write request to the kernel via the `write()` system call, our solution computes the entropy of the data that is to be written as a first step. Further actions are then taken based on the outcome of this computation. In the event that the entropy of the computed data exceeds a programmable threshold, the system proceeds to updating the `tgid_ransom_t` data structure (Listing IV.1) associated with the requesting process. At this point, the `tgid_ransom_t` of the requesting process is updated to reflect the total number of bytes that have been written using the `written_bytes` field. Similarly, we update

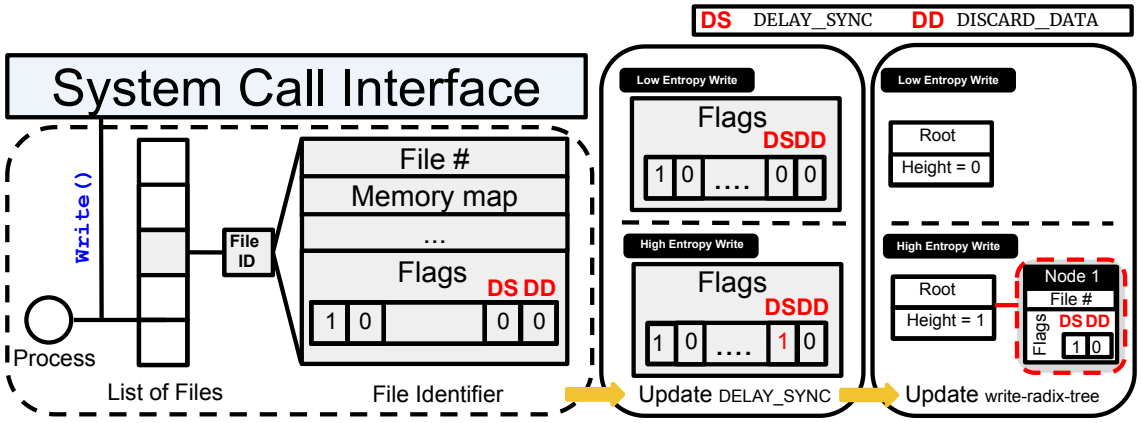
the information corresponding to the cumulative entropy of the observed data using the `cumulative_entropy` field. The `cumulative_entropy` and `written_bytes` fields are consumed later by the scheduler to compute the overall average entropy exhibited by the given process over a predefined execution period.

```
struct tgid_ransom_t {
    int periodic_cpu_time;
    int cumulative_entropy;
    int written_bytes;
    int socket_created;
    int delete_requested;
    int tcount;
};
```

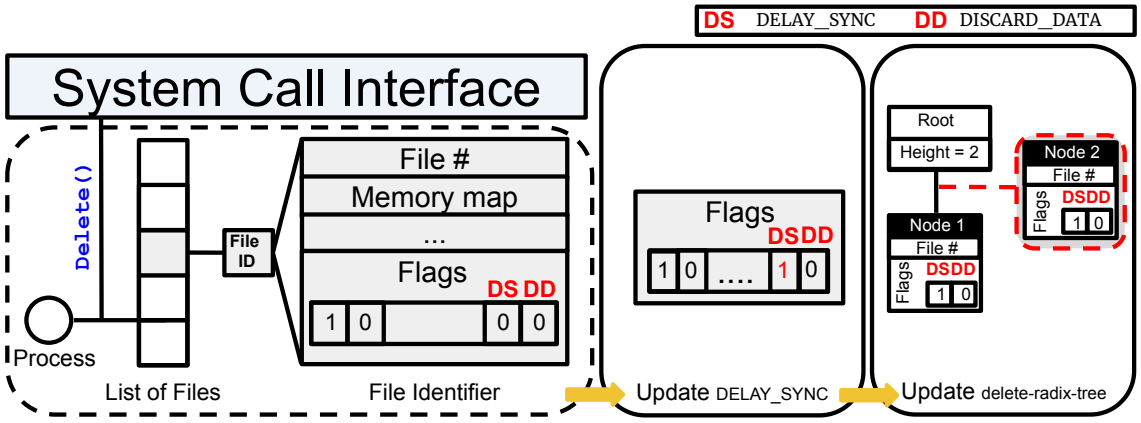
Listing IV.1: Data Structure for tracking a program's detection features.

In addition to computing the entropy, our design also tracks the files a given process has modified through write transactions. We accomplish this through the use of a radix tree. We choose this type of data structure in our design primarily because of its fast access time and ability to efficiently search data without the complexity of having to maintain a balanced tree. We create a `write-radix-tree` for each running process. Every time a process initiates high entropy writes to a file, an entry that points to the corresponding file's data structure is added to the tree.

In addition to tracking the affected files, we augment each entry within the tree with tags that denote specific actions. These actions are executed by the system anytime a tagged file is scheduled for synchronization to the backing store. To this end, write transactions that exhibit high entropy will result in the `DELAY_SYNC` tag being set. This tag informs the rest of the I/O subsystem to delay synchronizing the marked file to the disk until further assessment is made about its corresponding



(a) Write



(b) Delete

Figure 4.1: Overview of the system call interface and how different system calls update the tags of their radix trees.

process. In other words, the I/O activity issued to this file is still under evaluation by the defense system and could correlate to ransomware activity. An example that details this process is shown in Figure IV.1(a).

Delete Requests. Our solution enables the tracking of delete requests made by a running process. This step is necessary because not all ransomware perform in-place writes for encrypting user files. Most ransomware families create encrypted copies of the victim's files instead. Once such copies have been produced, the ransomware proceeds to deleting the victim's original data (files in plaintext). As such, our design tracks processes that perform delete operations. Whenever a process issues a

delete request to the kernel via the `delete()` system call, our solution updates a `delete_requested` field within the corresponding `tgid_ransom_t` data structure to reflect this behavior.

Our design aims to prevent the deletion of files in the event that the corresponding process is classified as ransomware. Therefore, in addition to postponing written data, our solution delays the deletion of files until a decision is made about the associated process. Since delete is an operation that is not frequently used by users, we add all the delete requests a given process initiates to a dedicated `delete-radix-tree`. Similar to the way our design handles write operations, every time a delete request is made by the running process, an entry that points to the corresponding file's data structure is added to the tree. Furthermore, all the delete requests that are recorded in the radix tree are marked with the `DELAY_SYNC` tag. This informs the system to not permanently delete the file from the backing store until further analysis is made. An example of this process is shown in Figure IV.1(b).

4.3.2 OS Scheduler

The OS scheduler represents a key component of our detection system. Its primary role entails periodically evaluating active processes on the system and classifying them as either benign, suspicious, or malicious. To this end, our scheduler relies on multiple features for classifying workloads. Such features include network requests, delete operations, and entropy measurements that are tracked by the system call interface. More specifically, the scheduler examines the `network_created` and `delete_requested` fields associated with each process to determine if network communication has been attempted and if any delete requests were issued. It also monitors the overall entropy associated with each process over a programmable execution window (e.g. 1 second). The scheduler employs this approach in order to validate that a process has sustained a sufficiently continuous stream of high entropy

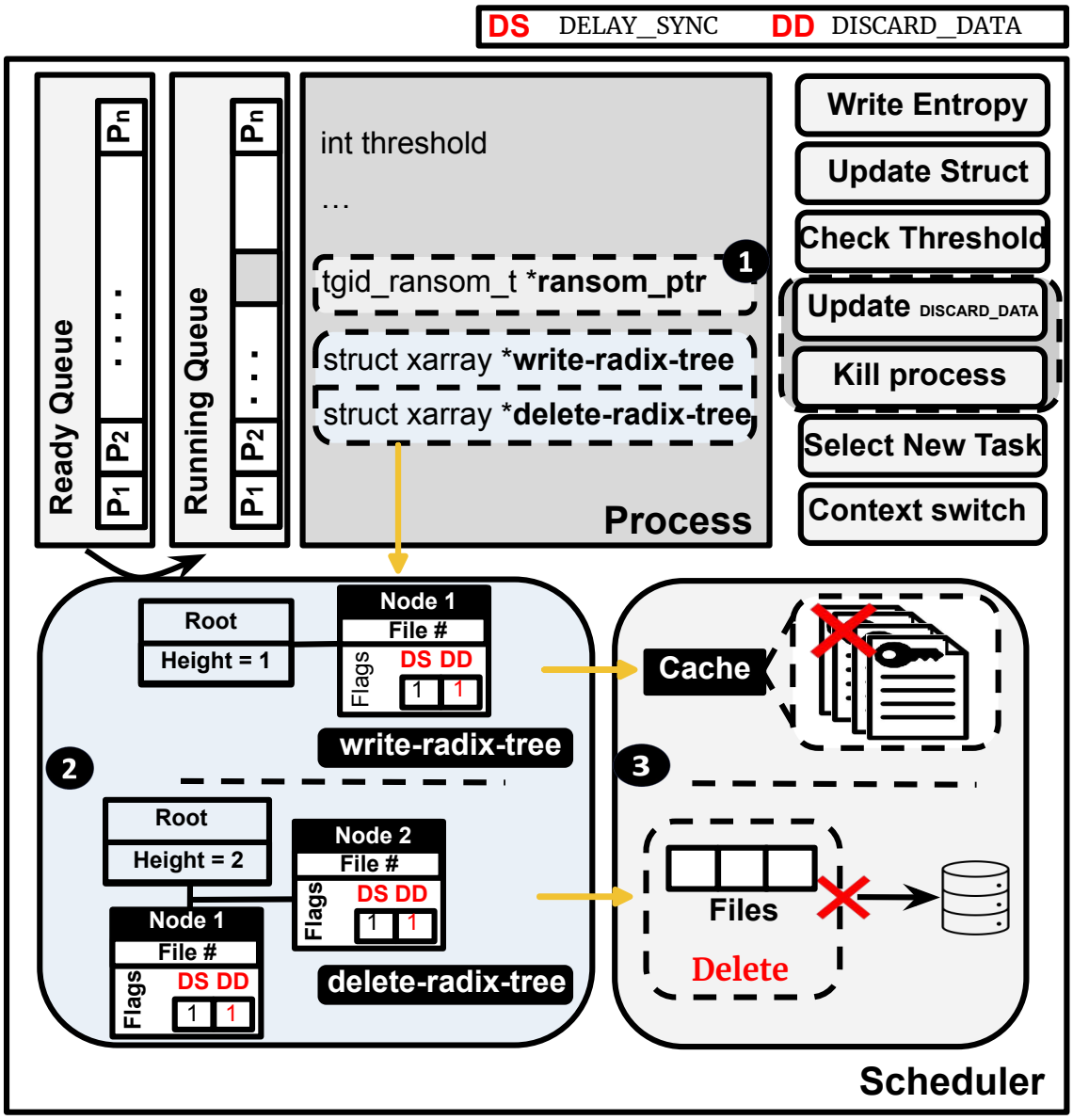


Figure 4.2: Overview of the OS scheduler design.

transactions before it considers the associated feature to be set. This method allows the detection system to mitigate the possibility of falsely misclassifying applications as a result of short-lived write transactions that may manifest high entropy levels.

The scheduler carries out a few tasks upon every context switch. This includes logging the amount of elapsed CPU time a given process was allocated on the system (using the `periodic_cpu_time` field). The scheduler uses this time to deter-

mine how often a process must be evaluated for its maliciousness. Once a process has executed for a predetermined period (e.g. 1 second), the scheduler references the `cumulative_entropy` and `written_bytes` fields that were previously saved by the system call interface and computes the average entropy of the write transactions the process has exhibited. If the computed average exceeds a predefined threshold, the process is considered to be suspicious (potentially malicious). Therefore, as an additional step, the scheduler checks the process's `socket_created` and `delete_requested` fields. If any of the aforementioned flags are set, in addition to the entropy exceeding a predefined threshold, the process is no longer considered to be suspicious, and is classified as malicious instead. Otherwise, the execution period (`periodic_cpu_time`), the cumulative entropy (`cumulative_entropy`), and the total number of written bytes (`written_bytes`) associated with the process are reset in preparation for a new evaluation cycle. The `socket_created` and `delete_requested` flags, on the other hand, are considered to be sticky. In other words, once set, they are not cleared throughout the lifetime of the process. An overview of this design is shown in Figure 4.2.

Discarding Malicious I/O Requests. Further action is taken by the scheduler whenever a process is deemed malicious. As previously stated, a process is classified as malicious if the entropy of its written data has exceeded a predefined threshold and at least one additional feature has been detected (network or file deletion request). Once a process is classified as malicious, the scheduler updates the tags of all previously marked files that are present in the process's radix trees. For instance, files that have been recorded in the `write-radix-tree` and `delete-radix-tree` structures are updated to include a `DISCARD_DATA` tag. Unlike the `DELAY_SYNC` tag that results in written data being postponed for synchronization, the `DISCARD_DATA` tag informs the I/O subsystem to discard data. It prompts the page cache to discard any memory pages that correlate to I/O data written by the malicious process. It also induces the

I/O subsystem to permanently discard any file deletion requests. The scheduler concludes by sending an alert to the user and consequently terminates the corresponding process. In the event that a process is only deemed suspicious (entropy threshold exceeded, but no other features detected), an alert is sent to inform the user about the abnormal activity. In addition, the user is prompted to either allow the process to continue running and be treated as a benign workload or terminate its execution and discard any delayed I/O requests. On the other hand, a process that is classified as benign (no features detected) results in its `write-radix-tree` being deleted. This in turn informs the page cache, that previously buffered write operations, can now be synchronized to the backing store. Furthermore, the design processes all of the previously buffered delete requests by permanently removing all of the files that have been tagged within the `delete-radix-tree`. The aforementioned tree is also destroyed once all of the delete requests have been successfully completed.

Detecting Multi-threaded Ransomware. Cybercriminals are constantly seeking ways to make their attacks more profitable. With this objective in mind, designing high performance programs that can outpace human response and evade detection systems is paramount. To this end, sophisticated forms of ransomware that pack multi-threaded support [115; 113; 71] have recently emerged into the malware landscape, posing a significant threat to both businesses and end users alike. Although this approach is typically aimed at defenses that rely on performance counters, it is conceivable that an attacker could harness this technique to evade our detection mechanism. For example, a malicious program could distribute its encryption activity across multiple threads such that the entropy of data written by each thread falls below our predefined threshold. To address such concerns, our system monitors the aggregated entropy that spans all write operations a given application issues. We achieve this by associating the `tgid_ransom_t` data structure with the program's main process (parent process) that each created thread (child process) can share.

The tgid_ransom_t structure shown in Listing IV.1 is designed for integration into different operating systems including popular ones, such as Linux and Windows. For instance, in Linux, threads that are spawned from a single program are assigned a common group ID (tgid). Our design uses this field as the basis for sharing the tgid_ransom_t structure across all downstream threads that share the same tgid. Since threads on this platform are also treated as processes, we augment Linux's standard process structure, task_struct, with an additional pointer (ransom_ptr). The scheduler and other parts of our design use this pointer for accessing tgid_ransom_t. Additionally, to ensure proper sharing of the tgid_ransom_t structure across the different threads associated with a given program, we modify the kernel's clone() system call. This enables our design to examine the corresponding tid of every newly created process before it is activated. If the new process shares the same tgid as its parent, then we simply set the ransom_ptr field in the task_struct to point to the parent's existing tgid_ransom_t structure. On the other hand, if the parent and the child have different tgid values, then a new tgid_ransom_t is allocated and initialized. Once a tgid_ransom_t structure has been allocated, all subsequent accesses to this resource are arbitrated for atomicity in order to prevent possible race conditions. Finally, our design keeps track of the number of threads that are referencing the tgid_ransom_t structure through a tcount field. This field is used to determine when the associated tgid_ransom_t structure for a given program needs to be deallocated. As such, whenever tcount is reduced to zero, the structure is deallocated implying that all of the program's threads have been terminated.

The Windows operating system uses a simpler support model for managing multi-threaded programs. On this platform, a multi-threaded application is conveniently represented by a single process by default. Windows uses a so-called Executive Process (EPROCESS data structure) for tracking such programs. Any subsequent threads that are instantiated from the executive process are allocated their own data struc-

tures (an ETHREAD structure for each thread). Although each ETHREAD is considered to be a separate structure, they are linked back to the program's executive process (EPROCESS). To this end, our design for Windows platforms entails adding the ransom_ptr field to the EPROCESS, which by default, is visible to all the associated ETHREADs. Similar to the Linux design, the ransom_ptr field would point to the same tgid_ransom_t shown in Listing IV.1. As such, whenever a given thread is dispatched, the Windows scheduler would simply examine the ETHREAD structure, point back to the ransom_ptr in the corresponding EPROCESS, and take the necessary action.

4.3.3 The Page Cache Subsystem

The page cache module is responsible for preventing malicious data from reaching the backing store. As a result, before this subsystem designates any of its pages for synchronization or eviction from its cache, it first determines if the associated file has been tagged. This prompts the system to look up the file and owning process of each memory page that is under consideration for commitment to the backing store. More specifically, the design uses a (file, process) tuple to determine if the file exists within the corresponding radix tree. If that file is found within the process's radix tree, then we proceed to examining the associated tags. On the other hand, if the file doesn't exist, we allow the respective pages to be committed.

In most cases, files that are written by user applications will not be recorded in the write-radix-tree since the entropy of such data is typically low. Under such circumstances, the ransom_ptr within the process's tgid_ransom_t would simply point to NULL implying that no radix tree exists. On the other hand, a file that exists within the radix tree and has the DELAY_SYNC tag set, as a result of write or delete operations, would result in the associated page to remain in the page cache until the scheduler classifies the associated process and updates the corresponding radix tree. In the event that the scheduler declares a process as malicious, the DISCARD_DATA tag

would be set. This in turn results in the associated page being freed and its entry removed from the cache without being committed to the backing store. In other words, the underlying file will retain its original content on the backing store and ignore any write or delete transactions initiated by ransomware. Therefore, the next time the file is opened by the user, the original data will be seamlessly mapped into memory without any impact.

Unfortunately, some ransomware families [112] can have adverse consequences on computer systems that are well beyond encrypting user files. Such behavior includes destructively overwriting the master boot record (MBR) in order to redirect the boot process to a malicious boot loader. This type of ransomware is difficult to detect because it postpones its encryption activity until the system is rebooted, giving full control to the malware. In other words, the encryption process does not start until the attacker's boot loader has overtaken the system. Since in this case, no high entropy would be detected due to the lack of encryption activity while the OS is running, our design additionally monitors write transactions that could impact the MBR. In most cases, the MBR, which corresponds to the first sector of a given storage device, is not overwritten by runtime applications. Instead, writes to this sector are typically restricted to OS installation activities. To address this issue, our design denies any access to the MBR while the OS is actively running. To achieve this, our solution examines the addresses of all blocks that are destined for storage devices. Any writes to the first sector of the device are denied followed by an alert being sent to the end user.

4.4 Methodology

We conducted ransomware experiments using the Cuckoo sandbox [116]. We chose this framework due to the various services it offers for testing malware, managing virtual machines, and performing analysis. We configured Cuckoo to run with a Windows

Family	Samples	Algorithm	Backup Spoliation	MBR	Multi-threaded	Defeated
7ev3n	9	RSA+AES-256	✓	✗	✗	✓
Cerber	46	RSA+RC4-256	✓	✗	✓	✓
Conti	57	RSA+AES-256	✓	✗	✓	✓
Crypmod	132	RSA+AES-256	✓	✗	✗	✓
Crypren	5	RSA+AES-256	✓	✗	✗	✓
CryptoWall	158	RSA+AES-256	✓	✗	✗	✓
DeriaLock	2	RSA+AES-256	✓	✗	✗	✓
Dharma	3	RSA+AES-256	✓	✗	✗	✓
InfinityCrypt	2	RSA+AES-256	✗	✗	✗	✓
Locky	351	RSA+AES-128	✓	✗	✗	✓
Maktub	3	RSA+AES-256	✓	✗	✗	✓
Rapid	4	RSA+AES-256	✗	✗	✗	✓
Petya	4	RSA+AES-256	✓	✓	✗	✓
RedEye	2	RSA+AES-256	✓	✓	✗	✓
Shade	13	RSA+AES-256	✓	✗	✗	✓
SporaRansomware	6	RSA+AES-256	✓	✗	✗	✓
TeslaCrypt	501	ECC+AES-256	✓	✗	✗	✓
WannaCry	26	RSA+AES-256	✓	✗	✗	✓
Total Samples	**1,324**					

Table 4.1: Summary of cryptographic ransomware families and their capabilities.

7 virtual machine that was launched on an Ubuntu 16.04 host. The Windows image was preloaded with commonly used files, such as, PDF documents, Word documents, Excel spreadsheets, and PowerPoint presentations, in addition to standard text files. We also provided regulated Internet access to the virtual machine through a filtered host-only adapter. This was setup to restrict any network activity to DNS, IRC, and HTTP traffic only. We allowed such basic networking activity purely for the purpose of enabling ransomware to communicate with their respective command and control (C&C) servers in order to facilitate key exchanges. For some ransomware samples, the encryption process was dependent on such exchanges before any malicious activity could be initiated.

We developed a proof of concept of our design using the Linux v5.10.4 kernel. This kernel was used to run the Ubuntu 16.04 host which in turn ran the Windows

7 image coupled with Cuckoo. We evaluated the robustness of our defense against 18 ransomware families. This resulted in the execution of 1324 real world samples, that were obtained from the VirusTotal [117] repository. Our dataset included both MBR and multi-threaded ransomware. The samples we used, their families, and respective capabilities are listed in Table 4.1. Each one of these samples was executed on the Windows 7 guest using Cuckoo for a minimum of 10 minutes or until files were encrypted. The Windows image was rolled back to a clean state every time a ransomware sample was executed. This was done in order to mitigate any lingering effects previously executed malware may have on future runs.

In addition to ransomware samples from VirusTotal, we executed 261 benign applications to evaluate the suitability of our design against false positives where each application was also executed for a period of 10 minutes. More specifically, we considered applications that spanned different categories including news, education, web browsing, social, communication, productivity, travel and local, health and fitness, and entertainment. This included the Office suite, video editing tools, compression utilities, and cryptographic applications to name a few. The aforementioned categories and their corresponding applications are summarized in Table 4.2.

In order to evaluate the robustness of our solution against false positives, we downloaded and installed commonly used applications for each category including productivity programs such as Office suite, Android Studio, and Eclipse that were configured to run on Linux. Each application was used interactively for at least 10 minutes while tracking the kernel alert messages. In addition, we launched common web services such as Google Maps, and TripAdvisor through Google Chrome. Furthermore, in order to evaluate our solution against applications that share ransomware-like features, we executed a variety of benign applications that span audio and video editing tools, compression utilities, and cryptographic applications.

We evaluated the overhead of our proof-of-concept using an HP laptop that was

Category	Applications	
Communication	*Beeper, Discord, Messenger, Flock, Geary, Google chats, ICQ,Skype, MailSpring, Mattercord, Microsoft Team, 4K Downloader,Telegram, Kontact, Konversation, Slack, , Whatsdesk, Yak Yak, Viewer, Pidgin, Zulip, Pixbuf, Signal, NitroShare*	
Education	*Genius, Grace, Klavaro, GLogic, Dictionary*	
Entertainment	*Amarok, Angry Birds, Collision, Clementine, Radar,Musiko,Lagno, klines, Kodi, kblocks, Kubrick, kapman, MPV, Moviesquare, VLC, Open Surge, Giraffe, PinTown, PlayOnLinux, Quadrapassel,Rili, Editor, SwellFoop, Tetravex, Chess,Google Play, SuperTuxKart, Spotify, Squarehead*	
Health & Fitness	*MyFitnessPal, WebMD, LiveStrong*	
News	*Akregator, Liferea, QuiteRSS*	
Productivity	*Android Studio, Viewer, Audacity, Bitwarden, SVG, Brackets, Tusk, Caffeine, Weather, Excel, Visio, PowerPoint, Word, Eclipse, Cura, Calibre, Carnet, Cheese, ClamTk, Converseen, Task Coach, Okular, Filezilla, FocusWriter, Font Manager, Geany, Gifex, GitKraken, Gscan2pdf, Handbrake, IDEA, Inkscape, Screenshot, Kdenlive, Krita, KolourPaint, Kontact, MATE, Picard, MyPaint, Nomacs, OBS studio, Omniawrite, Organizer, PDF, PDF2GO, Peek, Plume Creator, Qalculate, Rainbow, Scribus, Shutter, Planner, ScreenRecorder, SpeedCrunch, Finder, Sublime, Subtitld, Synaptic, Terminator, Calendar, Chartgeany, TeXstudio, Thunderbird, TurboWrap, Cleaner, Vim, VirtualBox, Zotero, Kazam, Todoist, Texi2pdf, Visual Studio, Dropbox, Ebook, WordPress, GLabels, GNOME, Wakeup,QtQR, Rambox,CopyQ, Everpad, Recorder*	
Social	*Twitter, Mumble, Somiibo*	
Travel & Local	*Airbnb, Google Maps, TripAdvisor, Uber, Yelb*	
Web Browser	*Arora, Chromium, Firefox, Google Chrome*	
Compression/Encryption	*pigz, xz, 7-zip, gzip, zip, lzop, lzma, pxz, bzip2, zstd, pax, cpio, iso, kgb, zpac, file roller, Lz4, plzip, pbzip2, p7zip,tar, ar,Winzip, WinRAR,peazip, sharl, bzip2, pixz,zpaq, zipx, AESCrypt, RSA*	
Photo/Video Editing	*Kdenlive, PiTiVi, OBS Studio, Shotcut, OpenShot, Cinelerra, Fotoxx, Pinta, ShowFoto, AfterShot, Photivo, UFRaw, GIMP, DIGIKAM, Adobe Photoshop, Pixeluvo, RawTherapee*	
Benchmark Suites	PARSEC-3	*Blackscholes, Bodytrack, Canneal, Dedup, Facesim, Ferret, X264, Fluidanimate, Freqmine, Raytrace, Streamcluster, Swaptions,Vips*
	SPLASH-3	*Barnes, Cholesky, Fft, Fmm, Lu, Ocean, Radiosity, Radix, Volrend, Water-nsquared, Water-spatial*
	SPEC2K6	*Astar, Bwaves, Bzip2, CactusADM, Calculix,Xalancbmk, Wrf, Gromacs, H264ref, Hmmer, Lbm, Leslie3d, Libquantum, Mcf, Perlbench, Povray, Sjeng, Soplex, Sphinx3, Tonto, Namd, Gcc, Gamess, GemsFDTD, Gobmk,Milc, Omnetpp, Zeusmp*

Table 4.2: Summary of benign applications and their respective categories.

equipped with an 8 core Intel Core i7 2670QM processor, 16 GB of memory, and 1 TB of storage. We ran a diverse set of realistic workloads commonly used for measuring performance directly on the Linux host. This entailed using a range of multi-threaded workloads from PARSEC [108] and SPLASH-3 [109] in order to characterize the design's sensitivity to compute-bound workloads. We also examined the overhead of our prototype in the context of I/O bound workloads through the Flexible I/O [110] and Filebench [111] benchmarks. This enabled us to systematically assess the performance overhead of our I/O subsystem when exposed to a large amount of sequential and random read/write operations on files and directories in the form of micro benchmarks. We also ran full fledged I/O centric applications (macro benchmarks) including various servers. The aforementioned benchmarks are listed in Table 4.3. The configuration of the respective I/O benchmarks is outlined in Tables 4.4 and 4.5.

Suite	**Benchmark**
PARSEC3	blackscholes, bodytrack, canneal, dedup, facesim, ferret, freqmine, fluidanimate, raytrace, streamcluster, swaptions, vips, x264
SPLASH3	barnes, cholesky, fft, fmm, lu, ocean, radiosity, radix, volrend, water_nsquared, water_spatial
Filebench (Macro)	varmail, fileserver, webserver, videoserver, webproxy, mongodb
Filebench (Micro)	openfiles, createfiles, copyfiles, deletefiles, listdirs, makedirs, removedirs
FIO	seqread, seqwrite, seqreadwrite, randomread, randomwrite, randomreadwrite

Table 4.3: Summary of performance workloads.

Workload	Configurations
seqread	4K sequential read, iodepth=1, numjobs=8
seqwrite	4K sequential write, iodepth=1, numjobs=8
seqrw	4K sequential rw (50%) each, iodepth=1, numjobs=8
randomread	4K random read, iodepth=1, numjobs=8
randomwrite	4K random write, iodepth=1, numjobs=8
randomrw	4K random rw (50%) each, iodepth=1, numjobs=8

Table 4.4: Configuration parameters of the FIO workloads.

Workload	Thread Count	File Count	File Size	R/W Size	Append Size
varmail	8	30K	16KB	1M	16KB
fileserver	8	10K	128KB	1M	16KB
webserver	8	50K	64KB	1M	8KB
videoserver	8	-	64KB	1M	-
webproxy	8	30K	16KB	1M	16KB
mongodb	8	10K	128KB	1M	16KB
openfiles	8	50K	-	1M	-
createfiles	8	30K	16KB	1M	-
copyfiles	8	10K	128KB	1M	-
deletefiles	8	50K	64KB	1M	-
listdirs	8	50K	-	1M	-
makedirs	8	50K	-	1M	-
removedirs	8	50K	-	1M	-

Table 4.5: Configuration of the Filebench workloads.

4.5 Evaluation

4.5.1 Ransomware Analysis

Feature Set Characterization. Our system relies on measuring the amount of entropy inherent in write operations as a primary detection feature. To this end, we conducted experiments across several programs in order to determine a suitable threshold for detecting ransomware activity. In particular, we characterized a total of 18 ransomware samples that belonged to different families (one sample from each family). In addition, we characterized several multi-threaded benchmarks from the

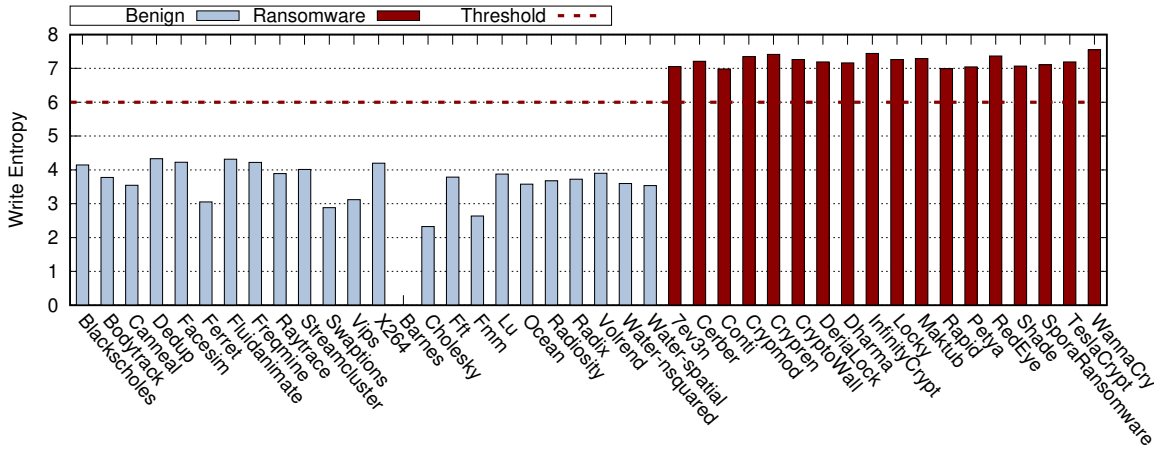


Figure 4.3: A comparison of write entropy measurements across PARSEC, SPLASH-3, and ransomware families over a one second execution period.

PARSEC 3.0 and SPLASH-3 suites. The entropy of each running program was measured over a period of one second. The results of this experiment are illustrated in Figure 4.3.

On average, we observed a relatively low entropy value across all benign workloads from the PARSEC and SPLASH-3 suites. These values ranged from 2.3 in the case of `cholesky`, to 4.3 while executing `fluidanimate`, with an overall average of 3.5. Furthermore, we found that multi-threaded programs, despite their low entropy, tended to issue write transactions that contained higher amounts of randomness when compared to single-threaded workloads. For instance, benchmarks from the SPEC2K6 [118] suite yielded a significantly lower average of 1.4.

On the other hand, we observed a drastic difference when comparing the aforementioned workloads to ransomware programs. The overall average across ransomware programs from the different families was 7.2. This figure correlates to a 2x increase relative to what was observed with benign programs. We found that the entropy of the tested malware ranged between 7 to 7.6 with ransomware from the `Cerber` family having the lowest value. To put things in perspective, `Cerber`'s entropy value was 60% higher than `fluidanimate`, a workload that had the highest entropy amongst

all benign applications. Based on such experiments, we concluded that using an entropy value of 6 represents a reliable threshold for distinguishing between benign and malicious activity.

In addition to entropy, our system treats network activity and file deletion requests as additional features for fingerprinting ransomware. Similar to the entropy experiment, we characterized a total of 18 ransomware samples that belonged to different families. Each sample was executed on an instrumented kernel that was designed to record high entropy writes, network activity, and file deletion requests, along with their respective timestamps. Overall, we found that all of the samples attempted to communicate with a C&C server after being launched on the system irrespective of whether the encryption key was generated locally or remotely. For example, although ransomware families, such as Conti, SporaRansomware, and WannaCry, are designed to receive their encryption keys from a C&C server, other families that produce their encryption keys directly on the victim's machine, including Cerber, Locky, and TeslaCrypt, still exchanged data over the network. We believe such traffic includes sharing the encryption key for data recovery purposes. In terms of delete requests, we observed that such operations were promptly issued after every newly encrypted file was generated. In other words, a file deletion request was issued to the filesystem immediately after an encrypted version of the associated file was produced. These results underscore the practicality of leveraging the aforementioned features for fingerprinting ransomware activity in addition to entropy measurements.

Detection and Filesystem Recovery. Our evaluation encompassed carrying out a large scale test effort in order to properly validate the ability of our design to detect and recover from ransomware. Our dataset consisted of more than one thousand ransomware samples and over 200 benign applications. The ransomware families and their respective families are shown in Table 4.1.

Overall, our results show that our solution is effective in dynamically distinguish-

Evaluation	*Results*
Total Samples	1585
Ransomware Samples	1324
Benign Applications	261
False Positives	0.0%
False Negatives	0.0%

Table 4.6: Summary of false positive and negative results.

ing between ransomware and benign activity. Our design was able to successfully detect all the malicious samples that we executed, including multi-threaded and MBR-based ransomware. Similarly, our solution demonstrated robustness against misclassifying benign applications as ransomware. Our solution was able to correctly classify all standard benign workloads without producing any false positives. For instance, programs such as Office suite, Visual Studio, and the Chrome browser were classified correctly without yielding any misclassification. In the case of compression and cryptographic utilities, the user was prompted to either approve or terminate the program. The results of this experiment are summarized in Table 4.6.

Our evaluation of the file recovery process entailed populating the system under test with several personal files that conformed to different formats including .pptx, .docx, and .txt. The filesystem had a total of 2700 files with different sizes ranging from 1KB to 22MB that were distributed across three folders: Documents, Downloads, and Desktop. Each directory contained approximately 2 GB worth of data. We observed that the majority of ransomware samples that we launched on the system would spend their first two minutes setting up their execution environment while attempting to communicate with a C&C server. The encryption process usually started immediately after ransomware had successfully obtained a key from its corresponding server. In all cases, our solution was able to detect such activity within one second of the encryption process. Upon detection, our system would restore all of the affected files within the aforementioned period, send an alert to the end user, and terminate

the offending process. In addition, we observed that some ransomware samples did not perform in-place writes of encrypted data. Instead, new copies were produced, encrypted, and then renamed to include a new extension. For instance, ransomware from the 7ev3n family added a .R5A extension while samples from InfinityCrypty appended a .enc extension. Once the newly created files have been produced on the system, the original files would then get deleted. Although our defense would simply leave the newly created files (with .R5A and .enc extensions) on the system, any attempt to delete the original files from the filesystem would be prevented through the use of a `delete-radix-tree`.

4.5.2 Performance Overhead

Compute-bound Workloads. Our runtime system has two main sources of overhead. The cost of context switching processes, and the cost of resource contention that stems from the execution of multiple threads. To this end, our evaluation focused on the use of multi-threaded benchmarks in order to accurately measure the overhead of our design on compute-bound workloads. The current version of our prototype implementation was geared towards collecting profiling information for the purpose of this study. Despite this, our solution is considered to be lightweight and incurs insignificant overhead.

Figure 4.4 summarizes the performance impact of our solution on compute-bound benchmarks from PARSEC and SPLASH-3. On average, we observed a performance reduction that was less than 2%. With the exception of a few programs, most benchmarks experienced an overhead that was well below this average. An exception to this trend, however, was `Ferret` which had an 11% reduction in performance. We attribute this overhead to the amount of context switches this process experienced relative to other workloads. For instance, `Ferret` had a 2x context switch rate relative to `Radix` which had the second highest overhead. Given that our scheduler requires

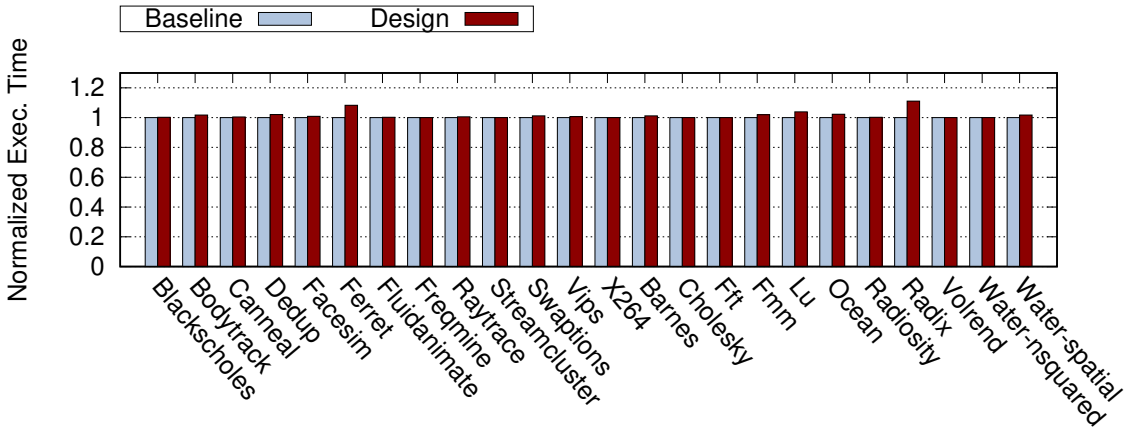
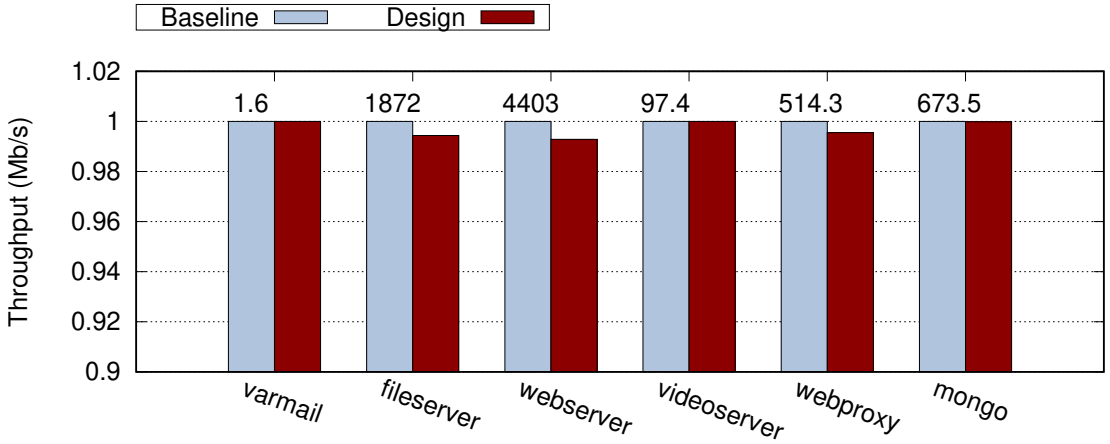


Figure 4.4: Performance impact of our design on compute-bound workloads relative to an unsecured baseline.

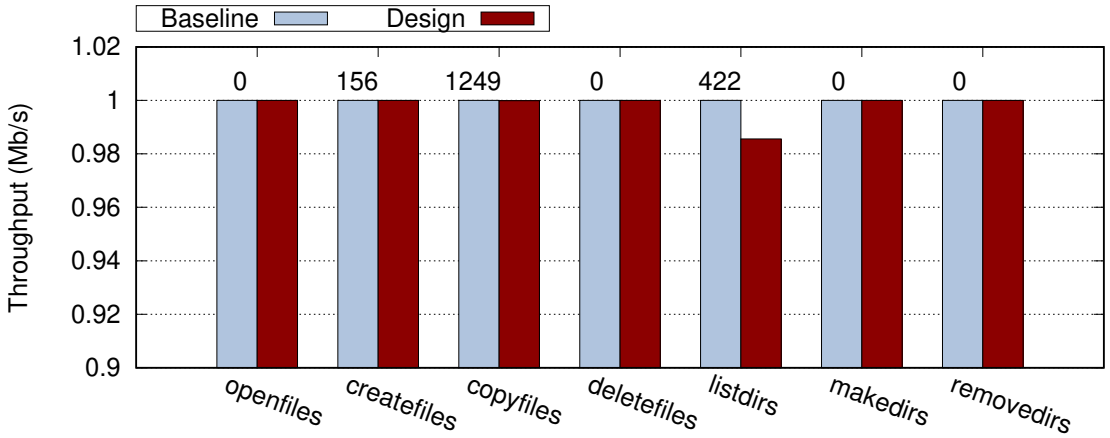
additional processing every time a program is context switched, having a process undergo excessive context switching will naturally lead to more overhead over time. `Radix`, a program that is designed to sort integers, exhibited similar delays, but to a lesser extent. Given that these workloads are calibrated to promote parallelism through multi-threading, we believe that the threads of these programs contended with one another over the shared `tigd_ransom_t` structure which contributed to an increase in the overall overhead. For example, we observed a relatively high amount of calls to locking constructs while executing the `Radix` workload.

I/O-bound Workloads. We conducted a variety of experiments that were designed to characterize the performance impact of our solution on I/O bound workloads. Our test coverage encompassed both macro and micro benchmarks from the Filebench and FIO suites. These suites allowed us to adequately exercise the I/O subsystem and measure its overhead in terms of both throughput and latency.

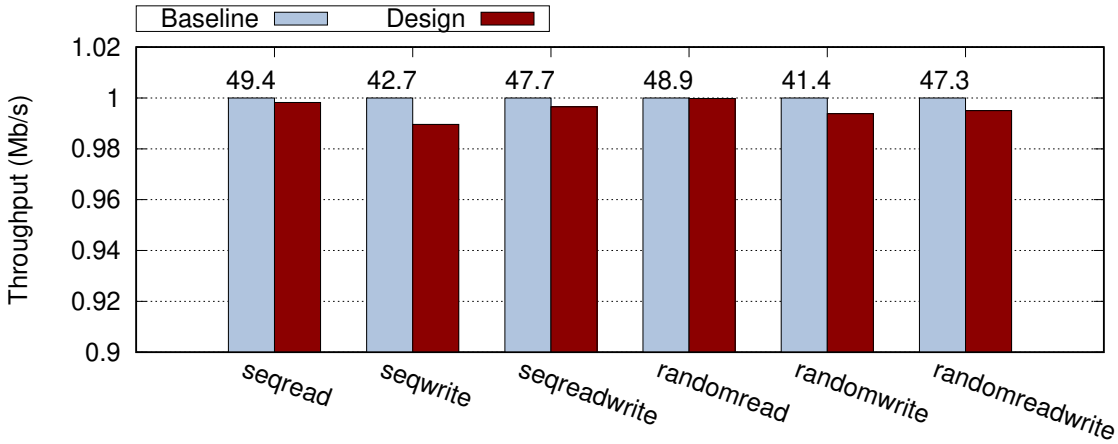
I/O Throughput. We first discuss the throughput of our solution and how it compares to an unsecured baseline design that uses a stock kernel. The results of this experiment are summarized in Figure 4.5. We observed that most of the applications from the Filebench suite experienced a slight decrease relative to the original design



(a) Filebench



(b) File operations

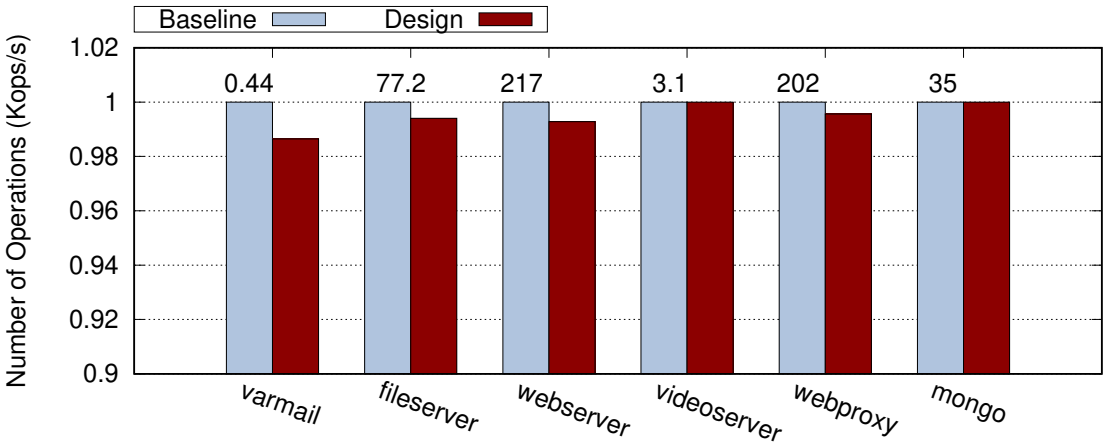


(c) FIO

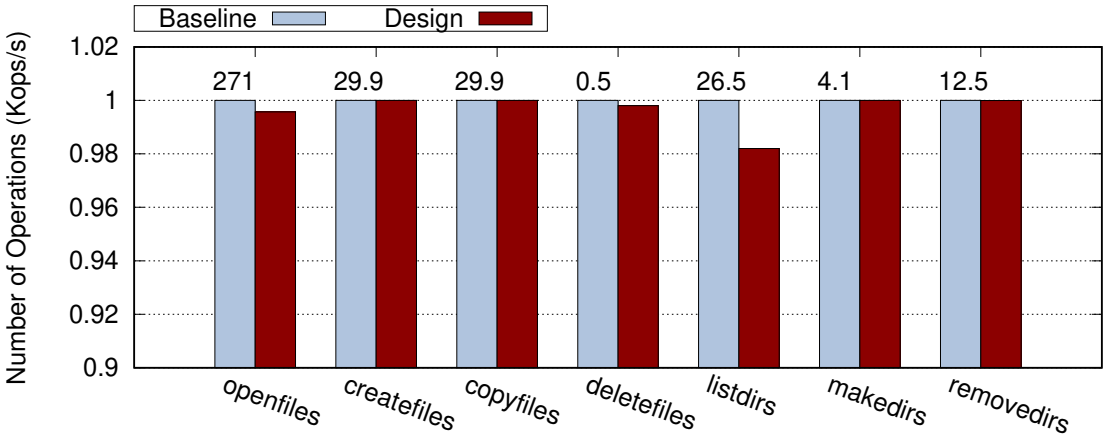
Figure 4.5: Summary of throughput experiments as a function of macro and micro benchmarks from Filebench and FIO.

after being configured with the parameters shown in Table 4.5. Although a slight decrease was recorded, the overall impact was minimal. On average, we observed a 6.4 Mb/s decrease in throughput across the various applications shown in Figure 4.5(a) which corresponds to a mere 0.2% reduction in performance. Overall, the `webserver` application experienced the most impact to throughput. This application observed a 31.6 Mb/s reduction in throughput relative to the baseline which started out with 1872 Mb/s. The `Filserver` application, on the other hand, experienced the second highest decrease after citing a 10.5 Mb/s reduction in throughput. We attribute this decrease to having more pages buffered in the page cache which in turn reduced the number of entries available for caching new data. This approach created unnecessary pressure on the overall page cache system which resulted in pages being evicted more often in order to accommodate newly fetched data. As a result, less I/O requests could be serviced out of the page cache and had to incur the additional penalty of fetching fresh pages from the backing store.

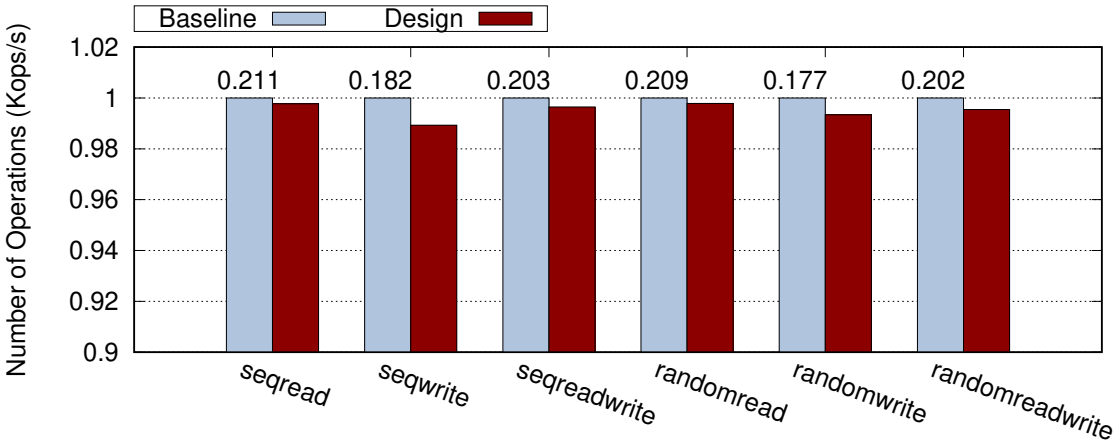
In addition to executing full fledged applications, we conducted various experiments that involved the use of microbenchmarks. We leveraged such benchmarks to exercise different file and metadata operations including `open`, `copy`, and `delete`, as well as commonly used directory operations. These are shown in Figure 4.5(b). In general, we observed a similar trend while testing the aforementioned operations. This trend was also seen when our design was evaluated against a series of sequential and random read/write operations. On average, our design observed a 12% reduction in throughput relative to the baseline with this reduction ranging between 0.5 Mb/s to 26.6 Mb/s. For the most part, we found that operations involving write transactions experienced the most reduction. For instance, operations such as random read and sequential read observed a mere 0.5 Mb/s and 5.2 Mb/s reduction, respectively. On the other hand, the remaining transactions that involved different write operations ranged between 10 Mb/s to 26.6 Mb/s. This is primarily due to the fact that



(a) Filebench



(b) File operations



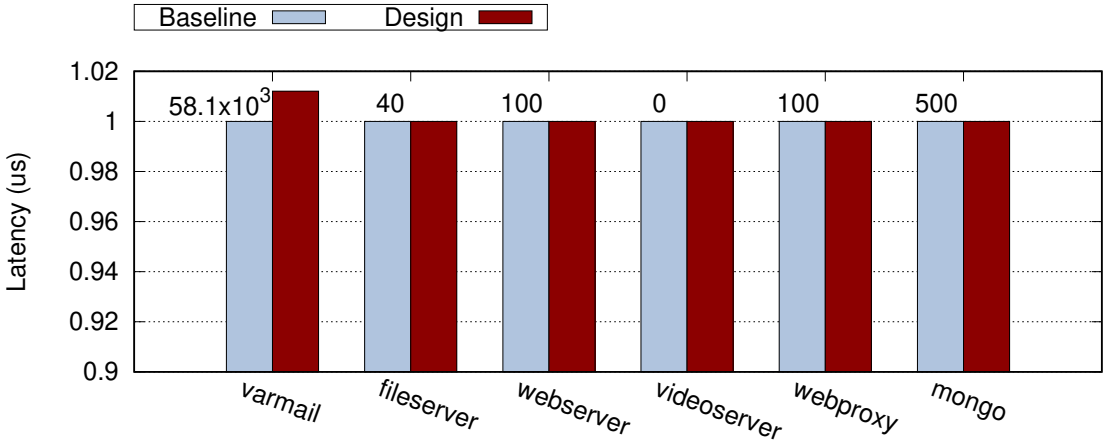
(c) FIO

Figure 4.6: Summary of IOPS experiments as a function of macro and micro benchmarks from Filebench and FIO.

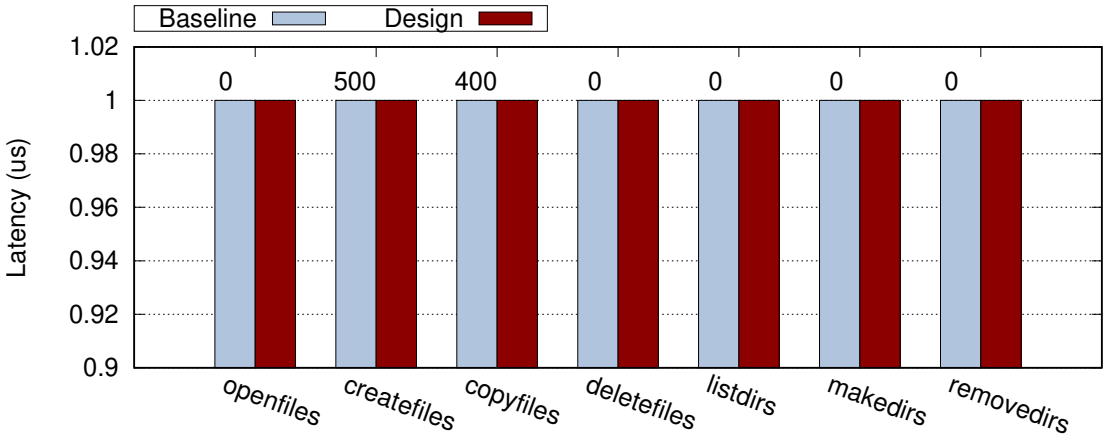
our system call interface performs additional steps on all write transactions that include entropy computations and radix tree updates. The aforementioned throughput results have almost a one-to-one correlation to data shown in Figure 4.6. In other words, similar trends can be derived when looking at this data from an I/O operations per second (IOPS) perspective.

I/O Latency. In addition to throughput, we evaluated the amount of latency our design introduces when accessing a single block of data. The results of this experiment are summarized in Figure 4.7. Similar to the previous throughput and IOPS experiments, we examined the impact of our design on latency across macro and micro benchmarks. From a macrobenchmark point of view, we found that `varmail` experienced the most overhead. This application had a 1.2% increase in latency relative to the baseline design. Also similar to previously reported throughput results, we found that operations that involved writing data tended to exhibit higher latencies. For instance, the latency of write operations observed increases that ranged between 0.4% to 1.1%. On the other hand, the overhead for read operations remained between 0.1% and 0.13%.

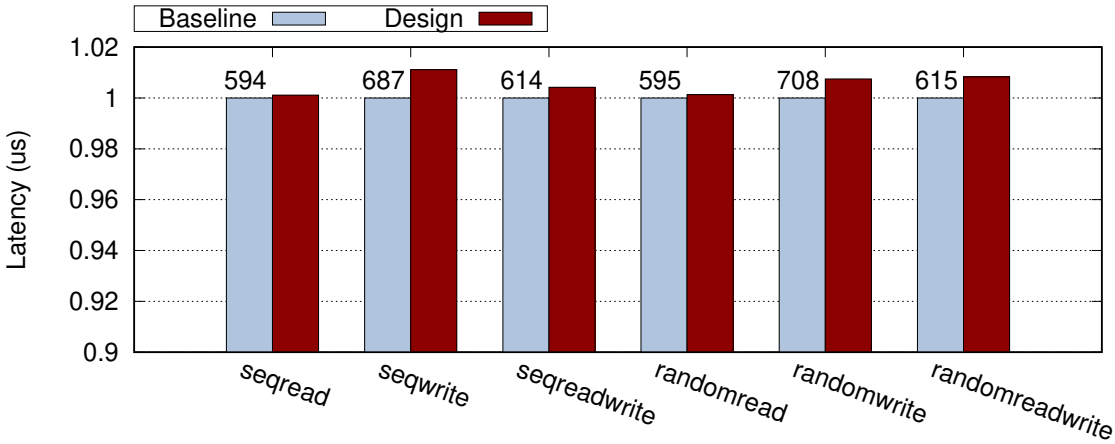
Overall, our results show that the performance impact to the I/O subsystem is minimal. Even when considering various metrics, our overhead remained well below the 1% mark. Such low overheads underscore the efficiency of our proposed approach.



(a) Filebench



(b) File operations



(c) FIO

Figure 4.7: Summary of latency experiments as a function of macro and micro benchmarks from Filebench and FIO.

CHAPTER V

Uncovering Corruption Ransomware: Confronting and Seamless Data Restoration

5.1 Research Objectives

In this chapter, we propose a new solution that can dynamically defend against corruption ransomware attacks. Unlike previous solutions that may lead to partially corrupted file systems or significant downtimes due to lengthy data recovery processes, our system guarantees the preservation of compromised data without requiring any manual recovery process or user intervention. The key observation of this work is that corruption ransomware first sends a copy of the victim's files through the network to a command and control (C&C) server before initiating the harmful data corruption process. Starting from this observation, we propose a design that combines network activity and filesystem activity monitoring with an efficient data synchronization management system between the memory and storage devices that protects the filesystem from being corrupted. We evaluate the effectiveness of our design against different data corruption tactics and show that our design effectively recovers all corrupted files. We also show the robustness of our solution against misclassifying benign applications as ransomware. We show that our proof-of-concept implementation has a minimal performance impact while executing a wide range of

workloads that are typically utilized to measure performance.

Generally, this chapter provides the following contributions:

- Proposes a dynamic solution that protects data from corruption ransomware attacks.
- Makes the observation that a combination of monitoring network and filesystem activity along with an effective synchronization management system for data between memory and storage devices can be utilized to safeguard data against corruption ransomware.
- Presents an end-to-end runtime solution that incurs negligible overhead across a wide range of standard benchmarks.
- Evaluates the effectiveness of our solution against different data corruption tactics and shows that it successfully recovers all affected files.
- Demonstrates the robustness of our design against misclassifying benign programs as ransomware.

The rest of this chapter is organized as follows: Section 5.2 details the threat model. Section 5.3 presents the design and implementation of the proposed defense system. Section 5.4 explains the experimental framework and Section 5.5 provides the evaluation results.

5.2 Threat Model

In this section, we present a detailed threat model and outline the assumptions we make about the ransomware actors and highlight their relevance to our solution. Our solution aims to protect computer systems from data corruption ransomware, which is designed to purposely corrupt the data of its victims and extort a ransom for

data restoration. Generally, we assume that ransomware typically infiltrates systems through various social engineering methods, such as phishing emails, social media platforms, and instant messaging applications. As an example, an attacker could implant the ransomware within a fraudulent email, with the intention of tricking victims into inadvertently downloading and executing it on their computer systems. We assume that ransomware has the ability to automatically scan the victim's filesystem, creating a list of files that match predefined extensions. We also assume that the created list of the victim's files is transmitted to the C&C server controlled by the attacker, where they are subsequently queued for the corruption process [68]. We also make the assumption that ransomware can corrupt the queued victim's files using the following strategies:

- **Data Swapping.** Ransomware can simply corrupt the victim's files by swapping the data between two randomly chosen files from the corruption queue [68].

- **Partial Encryption.** Ransomware can easily inflict corruption into the victim's files by partially encrypting the data of a randomly selected file from the corruption queue [69].

- **Data Distortion.** Ransomware can cause corruption into the victim's files by overwriting the data of a randomly chosen file from the corruption queue with null bytes, zeroes, or garbage values [68].

- **Data Deletion.** Ransomware has the capability to directly delete the queued victim's files after they are successfully transmitted and uploaded to the C&C server or produce corrupted copies of chosen files from the corruption queue and then delete the corresponding original files [70].

In addition to the above, we make the assumption that ransomware has the capability to employ various tactics during the corruption process, aiming to evade

detection. These techniques encompass deliberately corrupting data at lower I/O rates to decelerate the corruption process [72], and employing coincident execution via multi-threading to accelerate the corruption process [71]. In such tactics, our solution is designed to protect data from being corrupted. We also assume that the operating system is trustworthy and devoid of any privilege escalations. Otherwise, we argue that any defensive measures within the host, including anti-malware solutions, would be rendered ineffective. Thus, we presume that ransomware operates in user-mode.

5.3 The Corruption Ransomware Defense System

We propose a novel defense system that dynamically defends against data corruption ransomware attacks. Our solution guarantees the preservation of compromised data without the need for manual recovery or user intervention. The key observation of this work is that data corruption ransomware first sends a copy of the victim's files through the network to the C&C server under the attacker's control before initiating the harmful data corruption process. Based on this observation, we present a design that combines network and filesystem activity monitoring with an efficient data synchronization management system between memory and storage devices that protects the filesystem from being corrupted. To accomplish this, we devise our design to incorporate two innovative units, namely ransomware detection and data recovery. These units are tightly integrated with each other to ensure a cohesive defense system and seamless data recovery. An overview of the defense system's design is shown in Figure 5.1.

5.3.1 Ransomware Detection Unit

Our detection unit is designed to monitor the network and the filesystem activity carried out by an active process that has the potential to result in harmful corruption

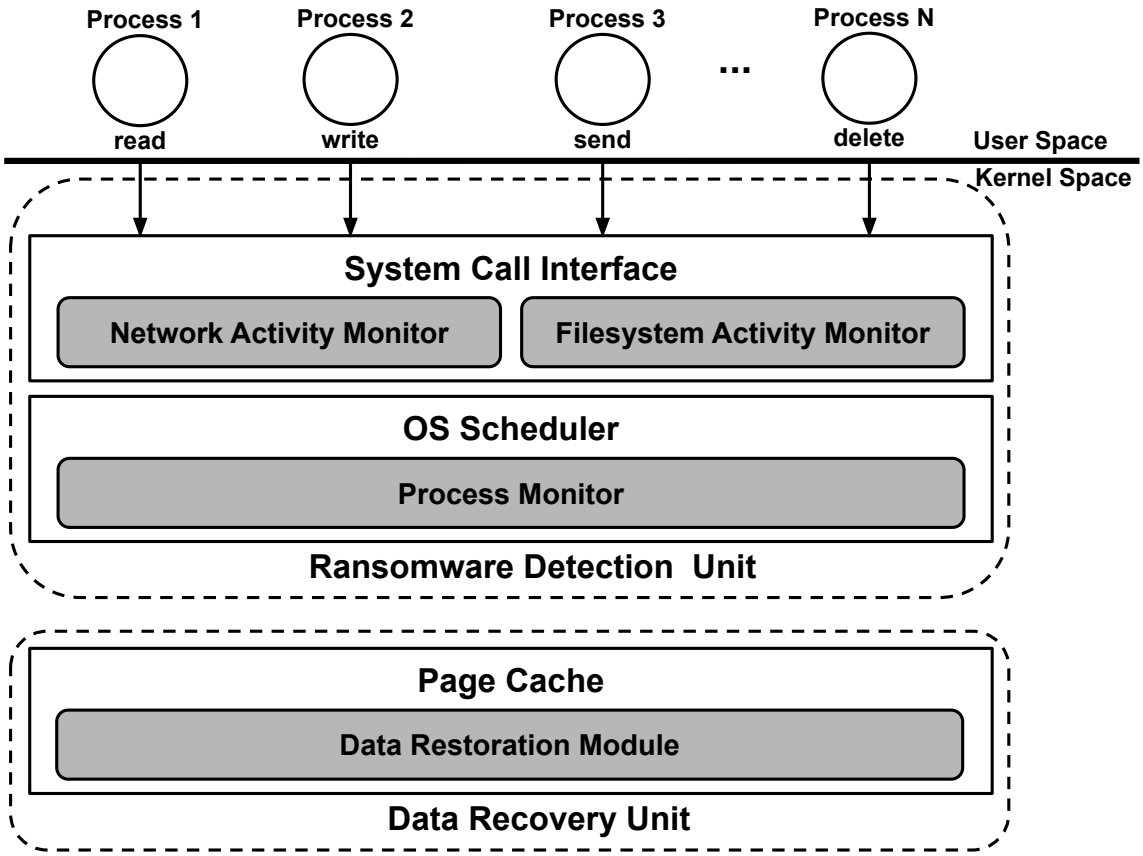


Figure 5.1: Overview of the ransomware defense system.

to the system files and prevent them to reach the underlying storage devices. To achieve this, we design our detection unit to comprise three novel modules: network activity monitoring module, filesystem activity monitoring module, and process monitoring module. These modules are tightly integrated with each other to ensure a cohesive detection system.

Network Activity Monitoring Module. An important characteristic of data corruption ransomware is that it sends a copy of the victim's files via a TCP connection through the network to ensure a reliable transmission to the C&C server controlled by the attacker [68]. This is typically accomplished by employing data streaming, utilizing `read()+write()/send()` system calls, or by transmitting the entire files through the `sendfile()` system call. As such, we augment our network activity monitoring module to track the files a given process has transmitted over the network through

`read()+write()/send()`, and `sendfile()` transactions. To accomplish this, we utilize a radix tree as part of our network activity monitoring module. We specifically chose this type of data structure in our network activity monitoring module due to its quick access time and efficient data search capabilities. We implement the network activity monitoring module inside the OS through changes to the system call interface. An overview of the network activity monitoring module is shown in Figure 5.2.

Our network activity monitoring module creates a `radix-tree` for every active process on the system. Whenever a process initiates a read request through the `read()` system call to the kernel, a reference to the relevant file is inserted into the radix tree. Subsequently, the module tracks transactions involving `write()` or `send()` operations. Whenever a process initiates a write request through the `write()` system call or a send request through the `send()` system call to the kernel, a reference to the relevant file is inserted into the radix tree. If the entry associated with a read request matches with the entry associated with a write request or a send request, the matched entry in the radix tree is marked with the `FILE_SENT_OVER_NETWORK` tag. This tag informs the system that the file associated with this entry is being transmitted via data streaming over the network to the C&C server, utilizing the combination of `read()+write()/send()` system calls.

In addition, our network activity monitoring module monitors transactions that involve `sendfile()` operations. Every time a process initiates a sendfile request through the `sendfile()` system call to the kernel, a reference to the relevant file is inserted into the radix tree and marked with the `FILE_SENT_OVER_NETWORK` tag. This tag serves as an indication to the system that the entire file associated with this entry is being transmitted over the network to the C&C server through the `sendfile()` system call. In the event that the file is sent over the network, the system subsequently updates the `tgid_ransom_t` data structure associated with the requesting process

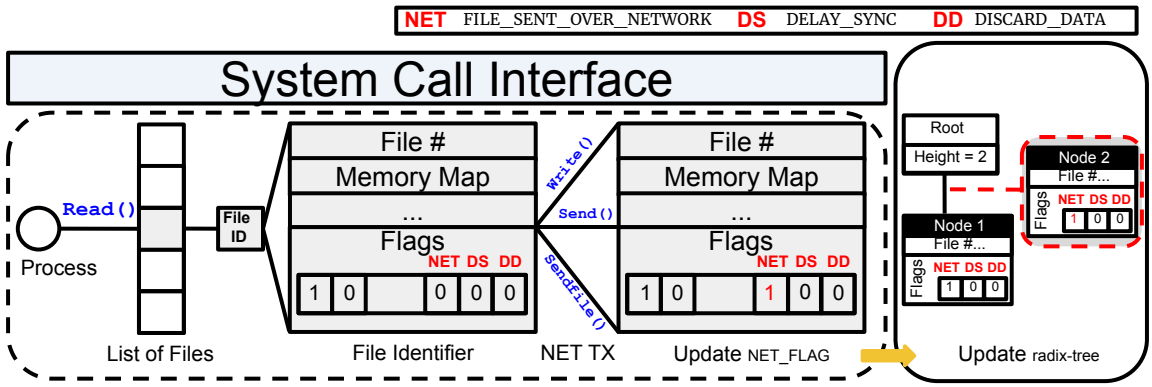


Figure 5.2: Overview of the network activity monitoring module.

as shown in Listing V.1. At this stage, the `file_sent_over_network` field of the requesting process's `tgid_ransom_t` is updated to represent the total number of files that have been transmitted over the network.

```
struct tgid_ransom_t {
    int cpu_runtime_period;
    int file_sent_over_network;
    int write_requested;
    int delete_requested;
    int tcount;
};
```

Listing V.1: Data structure utilized for monitoring the detection characteristics of a program.

Filesystem Activity Monitoring Module. Another important characteristic of data corruption ransomware is its practice of placing the victim's files that have been successfully transmitted and uploaded to the C&C server into a designated corruption queue. Subsequently, the files placed in the corruption queue undergo a corruption process in which they are overwritten with corrupted data [68; 69], or potentially deleted from the compromised system [70]. As such, we augment our filesystem

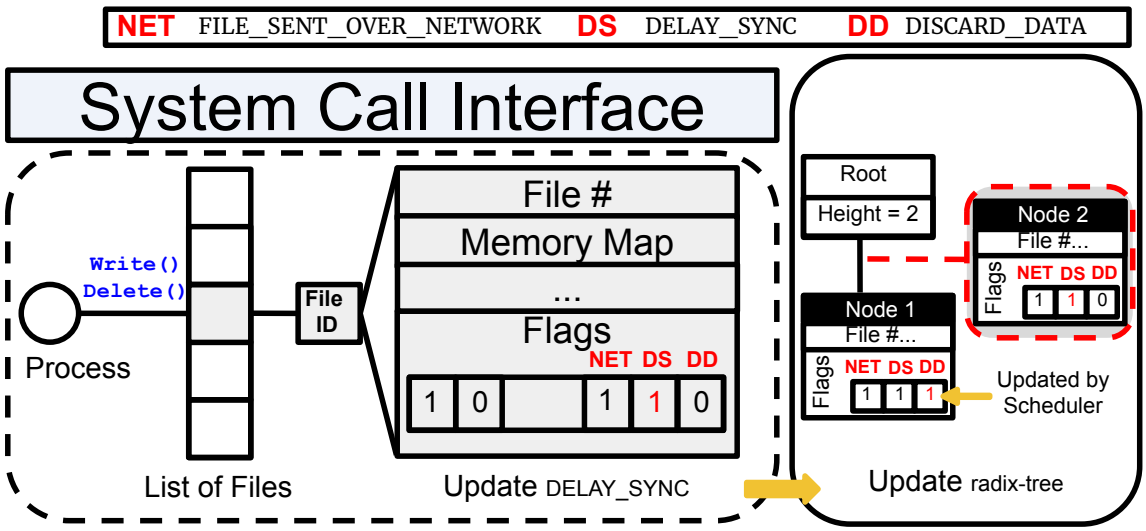


Figure 5.3: Overview of the filesystem activity monitoring module.

activity monitoring module to track processes that perform write and delete transactions. Similar to the way that our network activity monitoring module handles network transactions, we employ a radix tree within the filesystem activity monitoring module to monitor processes that perform write and delete transactions. We implement the filesystem activity monitoring module inside the OS through changes to the system call interface. An overview of the filesystem activity monitoring module is shown in Figure 5.3.

Our filesystem activity monitoring module tracks transactions that involve `write()` operations. Whenever a process initiates a write request through the `write()` system call to the kernel, an entry that refers to the corresponding file is inserted into the radix tree. If the entry associated with a write request is already marked with the `FILE_SENT_OVER_NETWORK` tag, it will then be marked with `DELAY_SYNC` tag. This tag serves to notify the system to postpone the synchronization of the marked file to the underlying storage devices until a more thorough evaluation of its associated process is conducted. In other words, the system is still evaluating the write operations performed on this file and there is a possibility of it being connected to ransomware activity. Subsequently, our filesystem activity monitoring module up-

dates the `write_requested` field in the corresponding `tgid_ransom_t` data structure to represent this behavior.

In addition, our filesystem activity monitoring module monitors transactions involving `delete()` operations. Every time a process initiates a delete request through the `delete()` system call to the kernel, an entry that refers to the corresponding file is inserted into the radix tree. If the entry associated with a delete request is already marked with the `FILE_SENT_OVER_NETWORK` tag, it will then be marked with `DELAY_SYNC` tag. This tag serves to notify the system to not permanently delete the file from the underlying storage devices until a more comprehensive assessment of its associated process is made. In other words, the system is still evaluating the delete operations performed on this file and there is a potential connection to ransomware activity. Subsequently, our filesystem activity monitoring module updates the `delete_requested` field within the associated `tgid_ransom_t` data structure to indicate this behavior.

Process Monitoring Module. The process monitoring module plays a crucial role in our detection unit, as it is tasked with periodically assessing active processes on the system, with the objective of classifying them as either benign or malicious. The process monitoring module relies on the analysis provided by the network and filesystem activity monitoring modules, with the objective of classifying the currently running processes on the system as either benign or malicious. In particular, the process monitoring module analyzes the `file_sent_over_network`, `write_requested`, and `delete_requested` fields associated with each active process to identify whether any files have been transmitted over the network and whether there have been any write or delete requests made to these files. We implement the process monitoring module inside the OS through changes to the OS scheduler. An overview of the process monitoring module is shown in Figure 5.4.

The process monitoring module carries out various tasks during each context

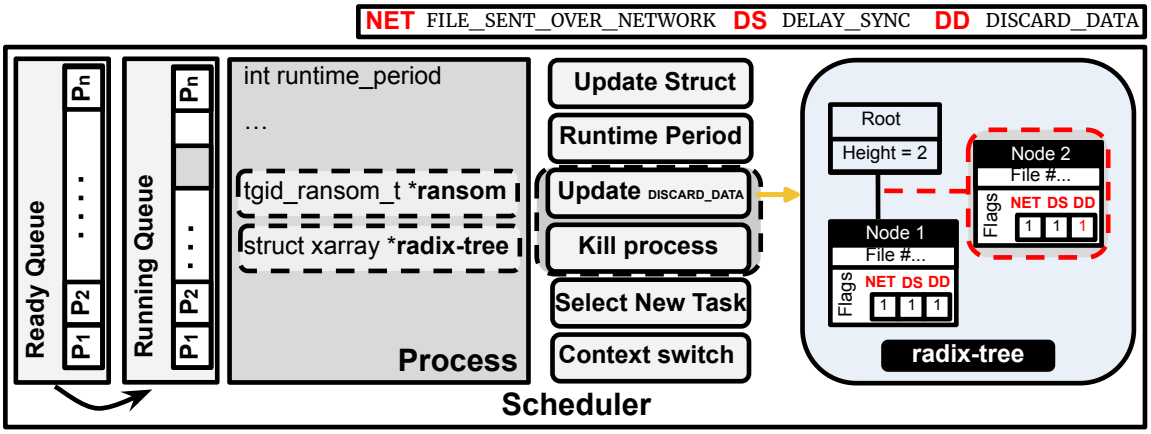


Figure 5.4: Overview of the process monitoring module.

switch. These tasks involve the CPU runtime period allocated to a given process on the system using the `cpu_runtime_period` field. The process monitoring module refers to this `cpu_runtime_period` field within the corresponding `tgid_ransom_t` data structure associated with the active process in order to determine how frequently the active process should be evaluated for malicious behavior. Once the active process has run for a predefined period (e.g. 1 second), the process monitoring module refers to the `file_sent_over_network`, `write_requested`, and `delete_requested` fields within the corresponding `tgid_ransom_t` data structure associated with the active process. The process monitoring module utilizes the stored fields from both the network activity monitoring module and the filesystem activity monitoring module to classify the processes running on the system as either benign or malicious. If the active process's `file_sent_over_network` tag is set, in addition to any of the `write_requested`, and `delete_requested` fields are set, the process is classified as malicious. Otherwise, the execution period `cpu_runtime_period` associated with the active process is reset in order to prepare for a new evaluation cycle.

Our detection unit relies on tracking the files a given process has transmitted over the network through the network activity monitoring module, in addition to tracking any write or delete requests made to these files through the filesystem activity

monitoring module. Where each process undergoes evaluation every predefined period (e.g. 1 second) through the process monitoring module to classify it as either benign or malicious. As such, our detection unit can effectively detect data corruption ransomware that employs malicious techniques such as a slowing tactic, which slowly corrupts the victim's data at low I/O rates in order to evade detection. Our detection unit can effectively detect such malicious techniques irrespective of the I/O rates employed by the ransomware, which can be as low as a single byte per second.

In addition, we augment our detection unit to detect data corruption ransomware that employs malicious techniques like a multi-threaded tactic, which distributes the corruption activities of ransomware across multiple cores, aiming to speed up the execution and avoid detection. To accomplish this, we associate the primary process (parent process) of the program with the corresponding `tgid_ransom_t` data structure. Consequently, all threads (child processes) created within the program can access and utilize the shared data structure. The threads created from a single program are typically assigned a common group ID (`tgid`). In our detection unit, we utilize this (`tgid`) field to enable the sharing of the `tgid_ransom_t` data structure among all created threads that have the same `tgid`. As threads are handled as processes, we create an additional pointer (`ransom_ptr`) in the process structure `task_struct` to access the `tgid_ransom_t` data structure. In order to facilitate the sharing of the `tgid_ransom_t` data structure among all program threads, we make modifications to the kernel's `clone()` system call, which involves examining the corresponding `tid` of each newly created process prior to its activation. If the new process and its parent process have the same `tgid`, we assign the `ransom_ptr` field in the `task_struct` to reference the `tgid_ransom_t` data structure of the parent process. However, If the new process and its parent process have different `tgid`, a new fresh `tgid_ransom_t` data structure is allocated and initialized. Once the allocation is complete, all subsequent accesses to this resource are carefully controlled to ensure atomicity and prevent

possible race conditions. Lastly, our detection unit maintains a count of the number of the active threads that reference the `tgid_ransom_t` data structure via a `tcount` field.

The process monitoring module takes additional actions once a process is classified. If a process is classified as malicious, the process monitoring module updates the tags of all the files that were previously marked in the process's radix tree by including a `DISCARD_DATA` tag. This tag serves as a signal to the data recovery unit in order to discard any malicious file data written or file deletion requests made by the malicious process. Our defense system then alerts the user and terminates the malicious process. On the contrary, if a process is classified as benign, the process monitoring module updates the data recovery unit in order to synchronize all previously buffered write transactions and delete requests present in the process's radix tree to the underlying storage devices.

5.3.2 Data Recovery Unit

The data recovery unit plays a vital role within our defense system, as it guarantees the preservation of compromised filesystem seamlessly without the need for any manual recovery process or user intervention. The data recovery unit comprises a novel component known as the seamless data restoration module, which is an essential component of our data recovery unit.

Seamless Data Restoration Module. The seamless data restoration module plays a crucial role in our data recovery unit, as it is responsible for restoring the compromised data seamlessly without the need for an explicit recovery procedure. The seamless data restoration module relies on the signal provided by the process monitoring module in order to efficiently manage the data synchronization between the memory and underlying storage devices, aiming to seamlessly protect the filesystem from the effects of the malicious ransomware process. To achieve this, we implement

the seamless data restoration module into the operating system by making modifications to the page cache. An overview of the seamless data restoration module is shown in Figure 5.5.

The primary role of the seamless data restoration module is to manage the data synchronization between the memory and the underlying storage devices, with the objective of preventing maliciously buffered data in the page cache from reaching the underlying storage devices. To this end, we augment the seamless data restoration module to verify the file that corresponds to the buffered data in the page cache before making a decision to either discard it from the page cache or sync it to the underlying storage devices. Initially, the seamless data restoration module verifies whether the corresponding file has been tagged. This action triggers the seamless data restoration module to search for both the file and its associated owning process in each memory page being considered for commitment to the underlying storage devices. In particular, the seamless data restoration module employs a (file, process) tuple to determine if the file is present in the process's radix tree. If the file is present in the process's radix tree, the next step involves analyzing the corresponding tags. On the contrary, if the file is not present in the process's radix tree, the respective pages are allowed to be committed without any additional consideration.

In general, files written or deleted by common user applications are not added to the process's `radix-tree` unless they have been transmitted over the network and followed by explicit write or delete requests. In such cases, the `ransom_ptr` in the process's `tgid_ransom_t` is set to `NULL`, indicating that there is no radix tree present. On the contrary, if a file is present in the process's radix tree and has the `DELAY_SYNC` flag marked due to write or delete transactions, the corresponding page remains in the page cache until the process monitoring module classifies the corresponding process and updates the radix tree accordingly. In case the process monitoring module states a process as malicious, this will result in the `DISCARD_DATA` tag being set. As a

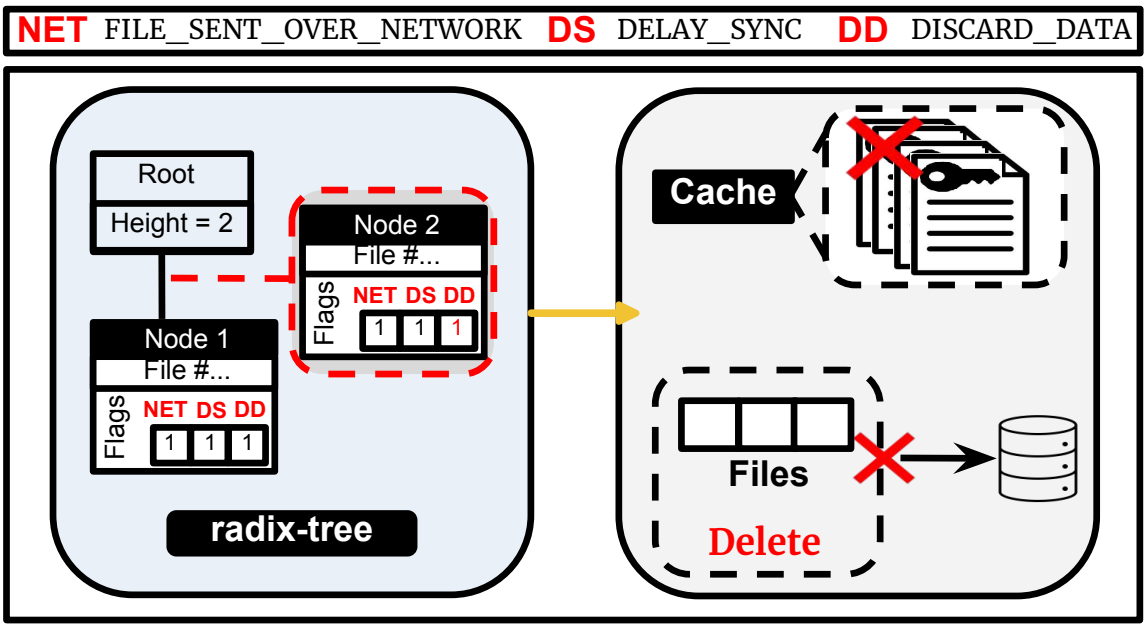


Figure 5.5: Overview of the data restoration module.

consequence, the seamless data restoration module permanently discards any file deletion requests, and it prompts the page cache to discard the corresponding page from the cache without synchronizing it with the underlying storage devices. In simple terms, the original content of the file on the underlying storage devices will remain unchanged, disregarding any write or delete operations initiated by the corruption ransomware. Hence, when the user reopens the file, the original content of the file will be seamlessly loaded into memory without any effect. However, if the process monitoring module declares a process as benign, the seamless data restoration module handles all the delete requests that were previously buffered and tagged within the `radix-tree` by permanently deleting them from the underlying storage devices. Additionally, the seamless data restoration module notifies the page cache that all the write transactions that were previously buffered and tagged within the `radix-tree` can now be synchronized to the underlying storage devices. Subsequently, the radix tree is ultimately destroyed.

5.4 Experimental Framework

We developed a proof of concept for our design by utilizing the Linux v5.10.4 kernel and Ubuntu 18.04. To evaluate the robustness of our defense system against data corruption ransomware attacks, we conducted different tests including data-swapping, partial encryption, data distortion, and data deletion. Each test case was executed employing two malicious techniques: slowing down I/O rates to one byte per second and utilizing eight threads for multi-threaded ransomware. In total, eight different test cases were performed. Each test was run for at least 10 minutes or until the files were corrupted. The operating system image was restored to a clean state prior to each test to ensure that any potential influences from previous tests were completely eliminated.

In addition to conducting eight distinct ransomware test cases, we also tested over 200 benign applications to assess the robustness of our design against false positives. These applications encompassed a wide range of categories including productivity, education, communication, web browsing, news, social media, health and fitness, travel and local services, and entertainment. This included Android Studio, Office Suite, web browsers, Eclipse, audio and video editing tools, TripAdvisor, Google Maps, compression applications, and cryptographic algorithms that were executed on Linux. Every application was used interactively for a period of 10 minutes, with close monitoring of kernel alert messages throughout the testing process.

We evaluated the performance overhead of our proof-of-concept utilizing an HP laptop, which had an Intel Core i7 2670QM processor (8 cores), 16 GB of RAM, and 1 TB of storage. We conducted a thorough assessment using a variety of standard benchmarks commonly utilized for measuring performance. To understand how our design performs under different computational demands, we executed a diverse set of workloads obtained from PARSEC [108] and SPLASH-3 [109] benchmarks. These workloads allowed us to analyze the sensitivity of our design to compute-bound tasks.

Additionally, we investigated how our design performed under different I/O bound workloads by executing a wide range of workloads obtained from the Flexible I/O [110] and the Filebench [111] benchmarks. These workloads allowed us to analyze the sensitivity of our design to I/O bound workloads, considering both micro and macro benchmarks. We also examined the performance impact of our solution in the context of network workloads using the Netperf [119] benchmarks. These workloads allowed us to analyze the performance overhead of our design when handling network-related tasks.

5.5 Evaluation

5.5.1 Detection and Data Recovery

In our evaluation, we conducted a diverse range of tests in order to adequately validate the capability of our solution in detecting and recovering from corruption ransomware attacks. Our evaluation encompassed eight distinct ransomware test cases along with more than 200 benign applications.

Overall, our results showed that our design was able to successfully distinguish between corruption ransomware and benign applications. Our solution effectively detected all eight executed ransomware test cases, including those that involved slowing down I/O rates and multi-threaded ransomware. Similarly, our results demonstrated that our design was robust against misclassifying benign workloads as corruption ransomware. Our solution successfully classified all executed benign applications without generating any false positives.

As part of our data recovery process evaluation, we populated the system with a diverse set of files that encompassed various extensions including .pptx, .docx, .xlsx, .pdf, and .txt. The filesystem contained a total of one thousand files, with sizes ranging from 1KB to 1MB. These files were distributed across different directories,

including Desktop, Downloads, and Documents. Our detection unit demonstrated the ability to detect all eight executed ransomware test cases within one second of the corruption process. Upon detection, our data recovery unit promptly restored all affected files by discarding any memory pages associated with the malicious process. It also ensured the protection of the filesystem by disregarding any file deletion requests from the malicious process. Subsequently, our defense system alerted the end user and terminated the offending process.

5.5.2 Performance Overhead

Compute-bound Benchmarks. Our evaluation focused on utilizing compute-bound workloads from PARSEC and SPLASH-3 in order to accurately measure the performance overhead of our design under different computational requirements. The performance impact of our solution on compute-bound benchmarks from PARSEC and SPLASH-3 is illustrated in Figure 5.6. On average, we observed a reduction in performance of less than 1%. The majority of benchmarks experienced an overhead lower than this average, with only a few exceptions. Notably, the `LU` and the `Dedup` workloads exhibited reductions in performance of 2.3% and 2%, respectively. The `LU` is a program designed to decompose a dense matrix into the product of an upper triangular matrix and a lower triangular matrix, while the `Dedup` is a program aimed to achieve high compression ratios by compressing data stream with a combination of global and local compression. Given that these workloads are specifically designed to promote parallelism through multi-threading, which leads to contention among the threads over the shared `tigd_ransom_t` structure, resulting in an overall increase in overhead.

Additionally, the `Ferret` workload also experienced a reduction in performance of 1.8%. The `Ferret` is a program designed for content-based similarity search on feature-rich data such as audio, images, video, and 3D shapes. That given the in-

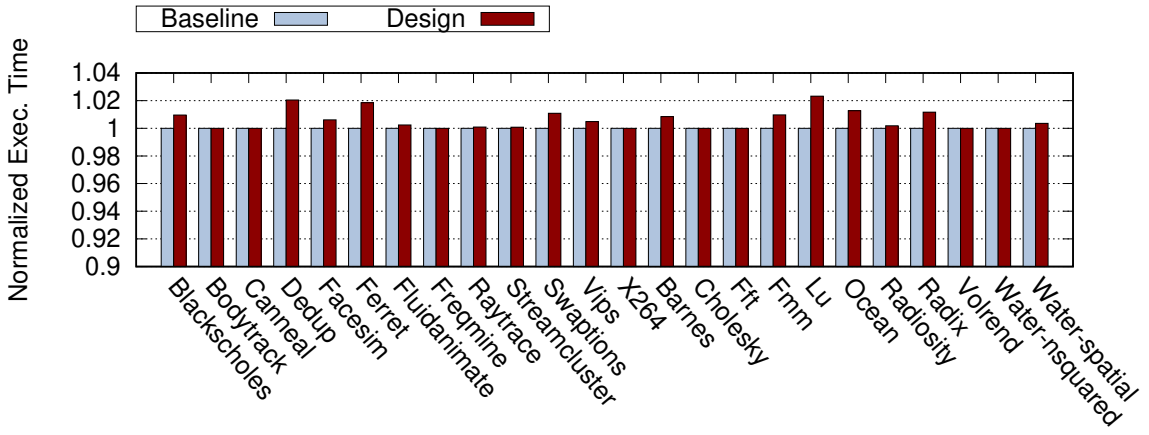


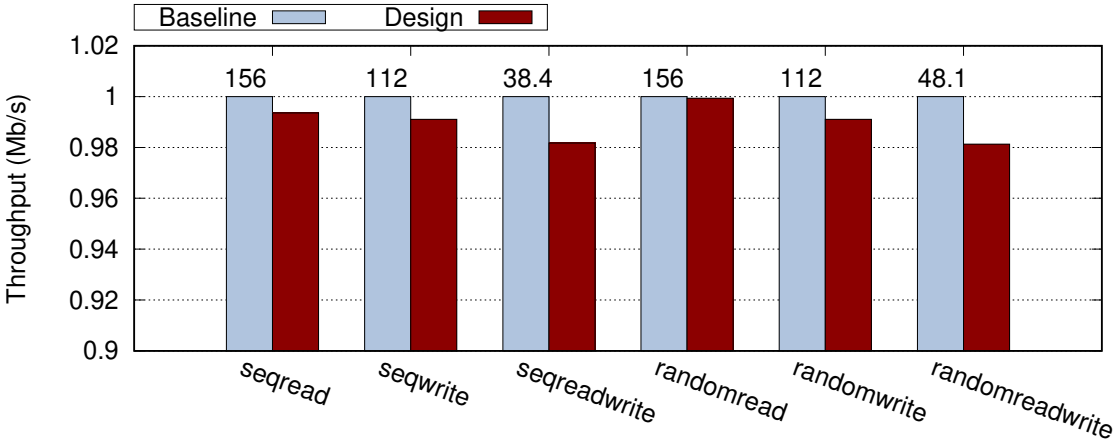
Figure 5.6: Performance overhead of our defense system on compute-bound benchmarks compared to a baseline system.

creased overhead observed for the `Ferret` workload can be attributed to its higher number of context switches compared to other workloads. As our process monitoring module requires additional processing during each context switch, a higher number of context switches naturally results in increased overhead over time.

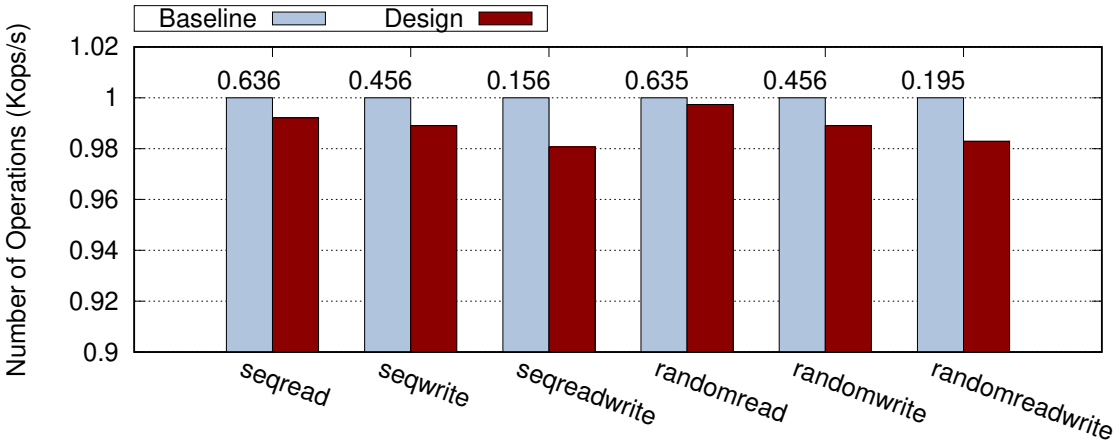
In general, our results demonstrated that the impact on compute-bound workloads was negligible, with an average impact of less than 1%. These low-performance costs showcase the effectiveness of our proposed design.

I/O-bound Benchmarks. We conducted a diverse set of experiments to evaluate how our solution performed under different I/O operations. Our evaluation focused on utilizing I/O bound workloads from the Fio and the Filebench benchmarks in order to adequately measure the performance impact of our design on such workloads. We took into account both micro and macro benchmarks to thoroughly measure the performance overhead of our solution in relation to I/O bound workloads.

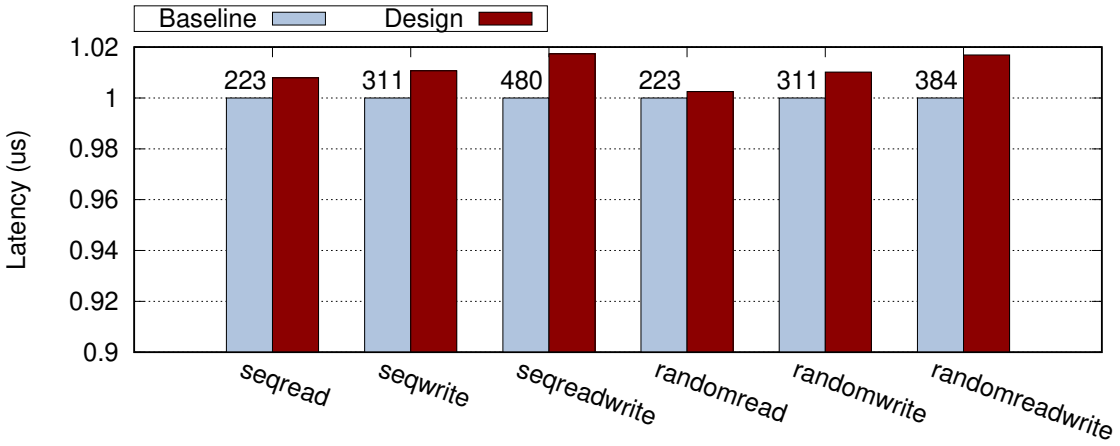
Micro Benchmarks. We conducted several experiments utilizing micro I/O bound workloads from the FIO and the Filebench suites in order to characterize the sensitivity of our design to such workloads. These benchmarks allowed us to thoroughly test the I/O subsystem and measure its performance impact in terms of throughput,



(a) Throughput



(b) IOPS



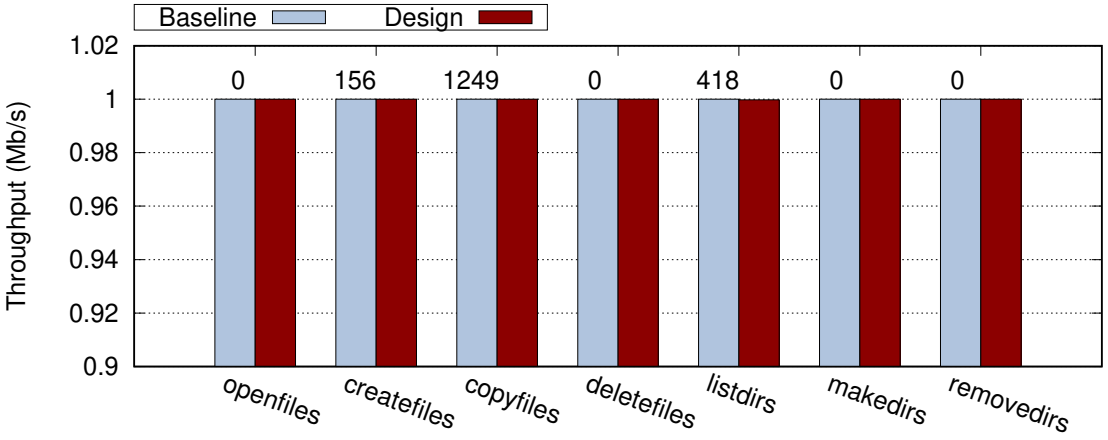
(c) Latency

Figure 5.7: Summary of micro experiments from FIO benchmarks as a function of throughput, IOPS, and latency.

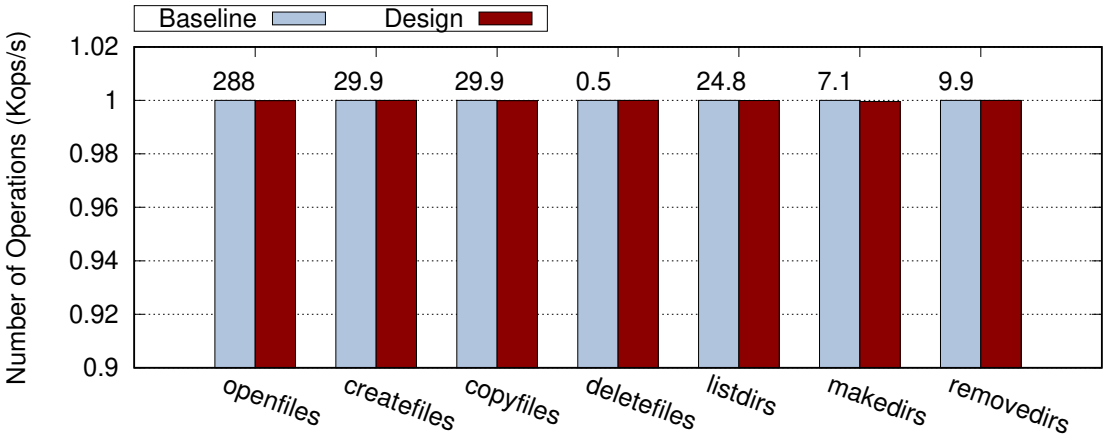
IOPS, and latency.

We utilized workloads from the FIO benchmarks in order to test the system under various file and metadata operations, such as sequential and random `read`, `write` operations. Figure 5.7 summarizes the micro experiments from FIO benchmarks as a function of throughput, IOPS, and latency. In terms of throughput, we observed that the majority of applications from the FIO suite exhibited a slight reduction in throughput to our defense system relative to the baseline system. On average, we noticed a decrease of 0.8 Mb/s in throughput for all applications depicted in Figure 5.7(a), which corresponds to a 1.0% decrease in performance compared to the baseline system. We also observed a similar trend when examining all the applications depicted in Figure 5.7(b) from the perspective of I/O operations per second (IOPS). In addition to evaluating throughput and IOPS, we also evaluated the impact of our design on latency for all applications from the FIO micro benchmarks, as demonstrated in Figure 5.7(c). We observed a slight increase in latency for read operations, ranging from 0.8% to 1.7%, as well as a slight increase in latency for write operations, ranging from 0.2% to 1.7%. We attribute this reduction in performance to the additional steps performed by our detection unit on all read and write operations, which involve radix tree updates.

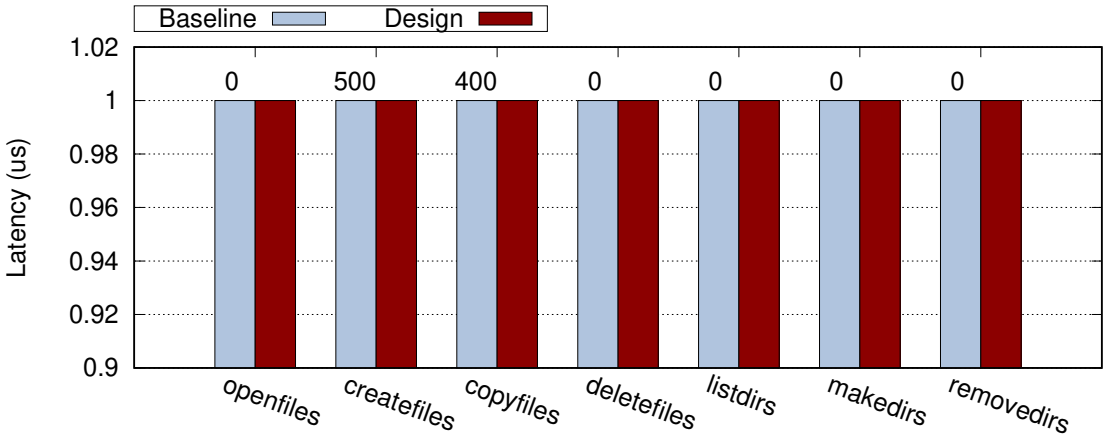
In addition to micro workloads from FIO benchmarks, we also conducted various experiments utilizing micro workloads from the Filebench suite. Such experiments include testing the system under various file and metadata operations, such as `open`, `copy`, and `delete` operations, in addition to commonly used directory operations. Figure 5.8 summarizes the micro experiments from Filbench benchmarks as a function of throughput, IOPS, and latency. We observed that our design had not any noticeable impact on micro workloads obtained from the Filebench benchmarks in terms of throughput, IOPS, and latency. This was anticipated as our design functionalities did not perform any significant additional steps to such files and directories operations.



(a) Throughput

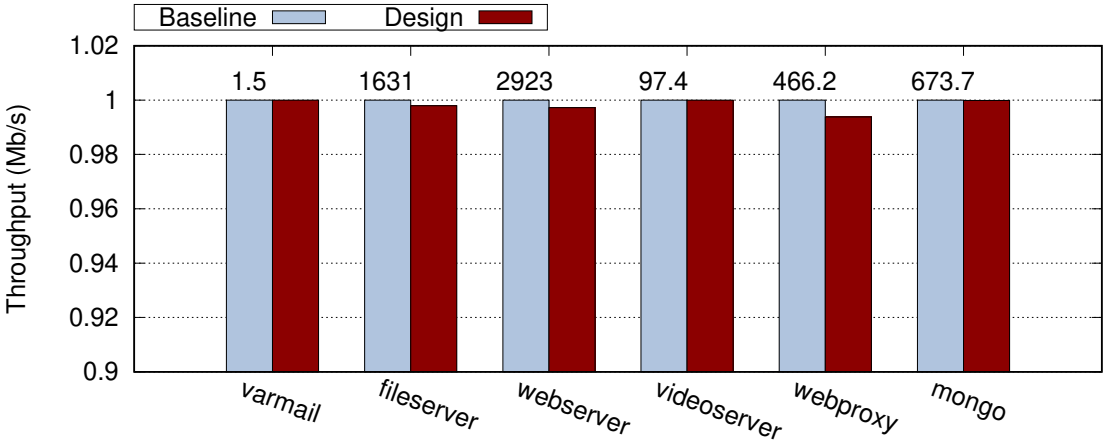


(b) IOPS

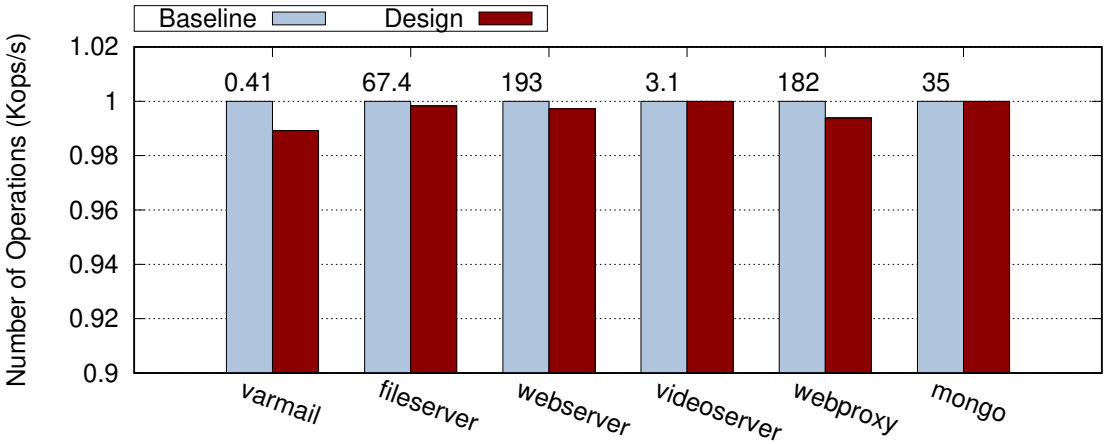


(c) Latency

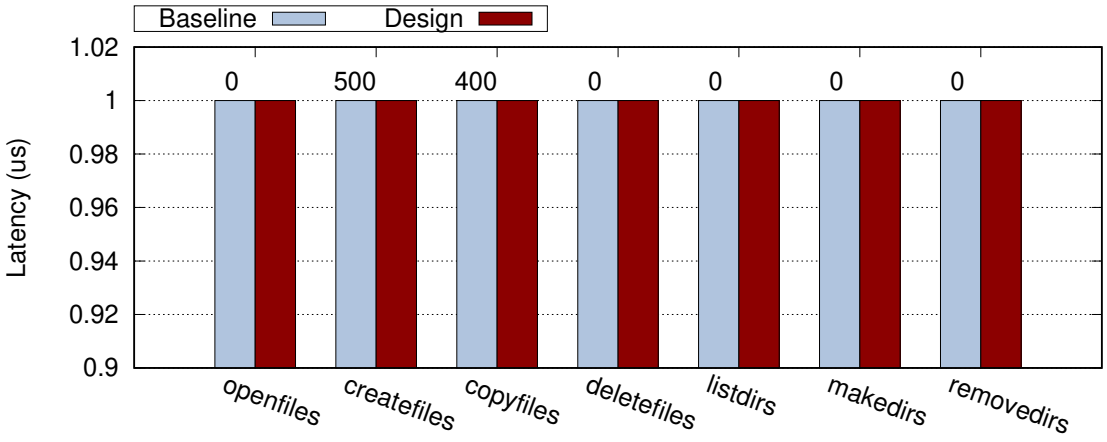
Figure 5.8: Summary of micro experiments from Filebench benchmarks as a function of throughput, IOPS, and latency.



(a) Throughput



(b) IOPS



(c) Latency

Figure 5.9: Summary of macro experiments from Filebench benchmarks as a function of throughput, IOPS, and latency.

Macro Benchmarks. In addition to micro benchmarks, we also conducted various experiments utilizing macro I/O bound workloads from the Filebench suite in order to characterize the sensitivity of our solution to such workloads. These workloads helped us to thoroughly test the I/O subsystem and assess its performance impact in terms of throughput, IOPS, and latency.

We employed workloads from the Filebench benchmarks in order to evaluate the system's performance across different large-scale operations such as `varmail`, `fileserver`, and `mongo`. Figure 5.9 summarizes the macro experiments from Filebench benchmarks as a function of throughput, IOPS, and latency. In terms of throughput, we observed that the `webproxy` application exhibited the highest level of overhead. This application experienced a reduction of 2.9 Mb/s in throughput, as depicted in Figure 5.9 (a), which corresponds to a minimal 0.6% reduction in performance relative to the baseline. From IOPS point of view, we observed that the `varmail` application experienced the highest level of overhead, as illustrated in Figure 5.9 (b). This application experienced a 1.0% reduction in the throughput compared to the baseline. In addition to evaluating throughput and IOPS, we also evaluated the impact of our design on latency for all applications from the Filebench macro benchmarks, as illustrated in Figure 5.9 (c). We found that our design had no noticeable impact on latency for macro workloads from the Filebench benchmarks.

Generally, our results indicate that the performance overhead on the I/O subsystem in the context of both micro and macro benchmarks was minimal, averaging less than 1%. Such minimal overheads demonstrate the efficiency of our proposed solution.

Network Benchmarks. We conducted various experiments through the use of the Netperf benchmarks in order to characterize the performance impact our design introduces when handling different network-related tasks. These benchmarks allowed us to adequately evaluate the performance overhead of our design in terms of throughput,

number of transactions per second, and latency.

We utilized workloads from the Netperf benchmarks in order to evaluate the system under various network tests. Such tests include data transmission test via `TCP_STREAM`, as well as data transmission test via `TCP_SENDFILE`, and network request-response test via `TCP_RR`. Figure 5.10 summarizes the Netperf experiments as a function of throughput, number of transactions per second, and latency.

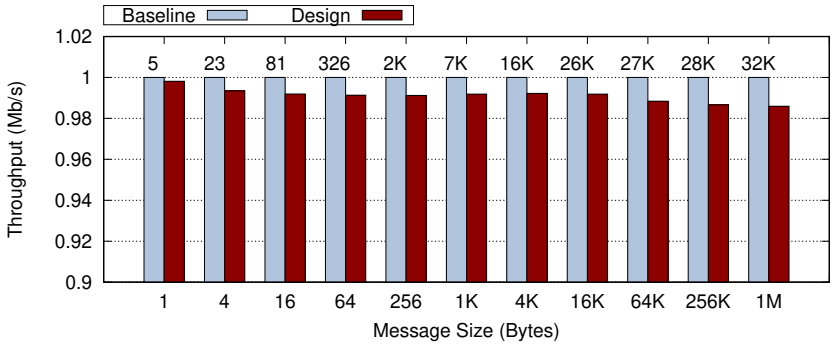
In terms of throughput, we performed measurements of throughput while varying the size of the transmitted message using the Netperf TCP Stream test. This test allowed us to gain valuable insights into how well the system handles data transmission via data streaming, specifically at different message sizes. The results of this test are illustrated in Figure 5.10 (a). We observed a slight reduction in throughput ranging from 0.2% to 1.4% while varying the size of the transmitted message from 1 byte to 1 MB. On average, we found a decrease of 142.3 Mb/s in throughput across all message sizes, which corresponds to a minimal 0.88% reduction in performance relative to the baseline system. Additionally, we also performed throughput measurements while varying the size of the transmitted message using the Netperf TCP Send File test. This test allowed as to have valuable insights into the system's ability to handle data transmission by sending entire files over the network, specifically considering different message sizes. The results of this test are demonstrated in Figure 5.10 (b). We observed a minor decline in throughput ranging from 0.3% to 1.4% while varying the size of the transmitted message from 1 byte to 1 MB. On average, we found a reduction of 224.1 Mb/s in throughput across all message sizes, which corresponds to a minimal 0.94% reduction in performance compared to the baseline system.

We also examined the impact of our design on the number of network transactions per second while varying the size of the transmitted message from 1 byte to 1 MB. Figure 5.10 (c) shows the number of transactions per second as a function of message size. We observed a slight decrease in the number of transactions per second falling

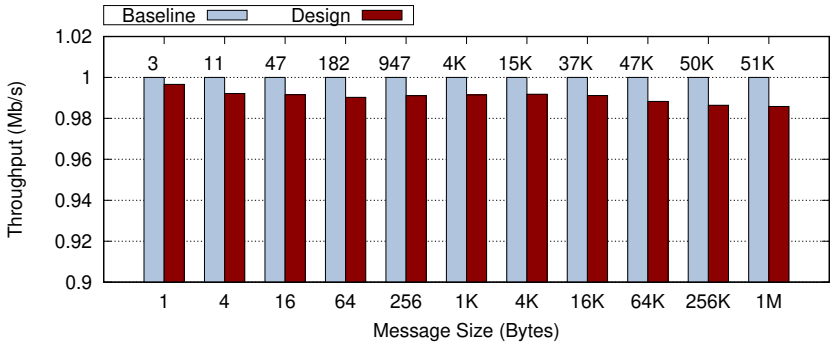
within the range of 0.4% to 1.7% while varying the size of the transmitted message from 1 byte to 1 MB. On average, we found a 180.9 Ktrans/s reduction in the number of network transactions across all message sizes, which corresponds to a minimal 0.78% decrease in performance relative to the baseline system.

In addition to throughput and number of network transactions per second, we also analyzed the impact of our design on latency while varying the size of the transmitted message from 1 byte to 1 MB. Figure 5.10 (d) shows the latency as a function of message size. We observed a slight increase in latency ranging from 0.4% to 1.0% as the size of the transmitted message varied from 1 byte to 1 MB. On average, we found a 0.8 μs increase in latency across all message sizes, which corresponds to a minimal 0.64% decrease in performance compared to the baseline system. In general, we found a reduction of performance of less than 1% our design introduces when handling different network-related tasks. We attribute this reduction in performance due to the fact that our network activity monitoring module performs additional steps on all network transactions, which involve radix tree updates.

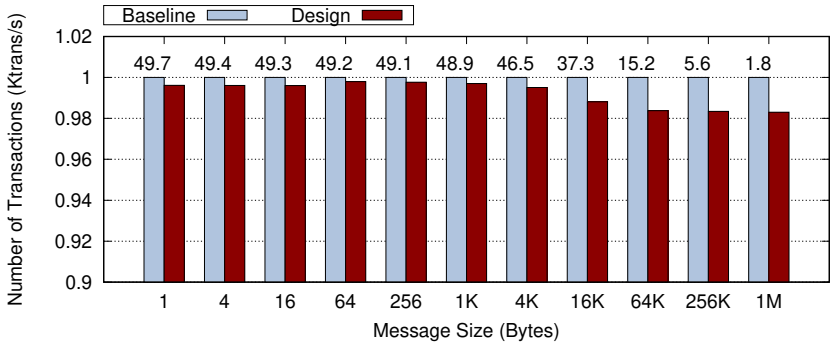
Overall, our findings indicate that the performance impact on network workloads was minimal, with an average decrease of less than 1%. Such minimal overheads demonstrate the effectiveness of our proposed design.



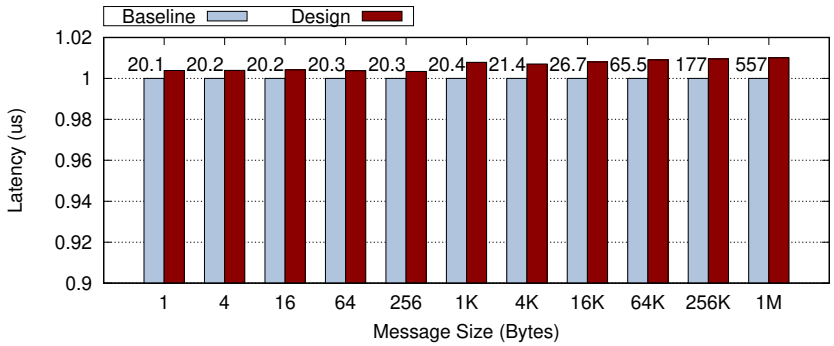
(a) Throughput vs Message Size (TCP Stream)



(b) Throughput vs Message Size (TCP Send File)



(c) Number of Transactions Per Second vs Message Size



(d) Latency vs Message Size

Figure 5.10: Summary of Netperf experiments as a function of throughput, number of transactions per second, and latency.

CHAPTER VI

Sniper: Countering Locker Ransomware Attacks Through Natural Language Processing

6.1 Research Objectives

In this chapter, we present a novel runtime system that dynamically restores device access by undoing the effects of locker ransomware. Unlike prior work, our solution seamlessly safeguards compromised devices from such attacks without the need for a manual recovery process or user intervention. A key observation made by this work is that attackers rely on the display of a ransom note on the victim's device to demand payment. Starting from this observation, we propose a design that combines the monitoring of mobile app activity with a natural language processing (NLP) unit that harnesses transformers to detect the appearance of ransom notes. We thoroughly evaluate the robustness of our solution against more than 5K ransomware samples that span more than 17 families. We show that our solution reliably recovers from all the malicious samples that we tested, including overlay screen and change pin-code ransomware. We show that our experimental proof-of-concept exhibits minimal performance impact while running a mix of mobile benchmark applications [120].

Overall, this chapter makes the following contributions:

- Proposes a novel runtime defense that protects mobile devices from the effects

of locker ransomware.

- Makes the observation that combining natural language processing with transformers can be harnessed to detect the appearance of ransom notes and consequently identify the presence of locker ransomware.

- Discusses the impact of adversarial machine learning while demonstrating the robustness of our solution against such attacks.

- Presents an end-to-end runtime solution that incurs negligible overhead across a wide range of standard benchmarks.

- Characterizes the resiliency of our design against more than 5K ransomware samples and shows that our solution robustly recovers from the effects of all the samples we tested.

The rest of this chapter is organized as follows: Section 6.2 discusses the threat model. Section 6.3 illustrates the design of the proposed defense system. Section 6.4 presents the methodology and experimental framework used in this work and Section 6.5 discusses the results of our evaluation.

6.2 Threat Model

In this section we detail the threat model and the assumptions we make about the attackers and how it applies to our design. Our solution is designed to protect mobile systems against ransomware that deliberately locks victims out of their devices. In general, we make the assumption that ransomware is installed through the proper mechanisms already supported by the device's platform. For instance, an attacker can publish seemingly benign .apk files on a mobile application provider, such as Google Play Store that victims unwittingly install onto their devices. Overall, we

make the assumption that any of the following ransomware types can be launched on a system:

- **Overlay Screen Ransomware.** This type of malware is assumed to have the ability to pop a new window that overlays on top of other apps, and consequently, prevent the victim from utilizing their device [121]. Attackers generally rely on social engineering techniques, such as phishing emails and websites, social media, torrent sites, instant messaging apps, and third-party app stores to infect a victim's device with ransomware.

- **Lock Screen and Pin Code Ransomware.** This type of ransomware is assumed to have the ability to change the pin code of a victim's lock screen, and consequently, deny access to the device. Akin to the aforementioned category, attackers rely on social engineering strategies for infecting users [122].

In addition to the above, we make the assumption that the attacker is aware of the architecture of our NLP module, and is therefore, capable of conducting adversarial text attacks to evade detection. Another possible attack involves physical access to a device. This method, however, contradicts an important objective ransomware aims to achieve. Ransomware campaigns are often concerned with compromising as many users as possible in order to remain profitable. As such, we do not envision physical attacks being a practical approach in this case.

6.3 The Sniper Defense System

We propose Sniper, a novel system that dynamically restores device access by undoing the effects of locker ransomware. A key observation made by this work is that attackers rely on the display of a prominent ransom note on the victim's device to demand payment. Starting from this observation, we devise a solution that combines the monitoring of mobile app activity with a natural language processing (NLP) unit.

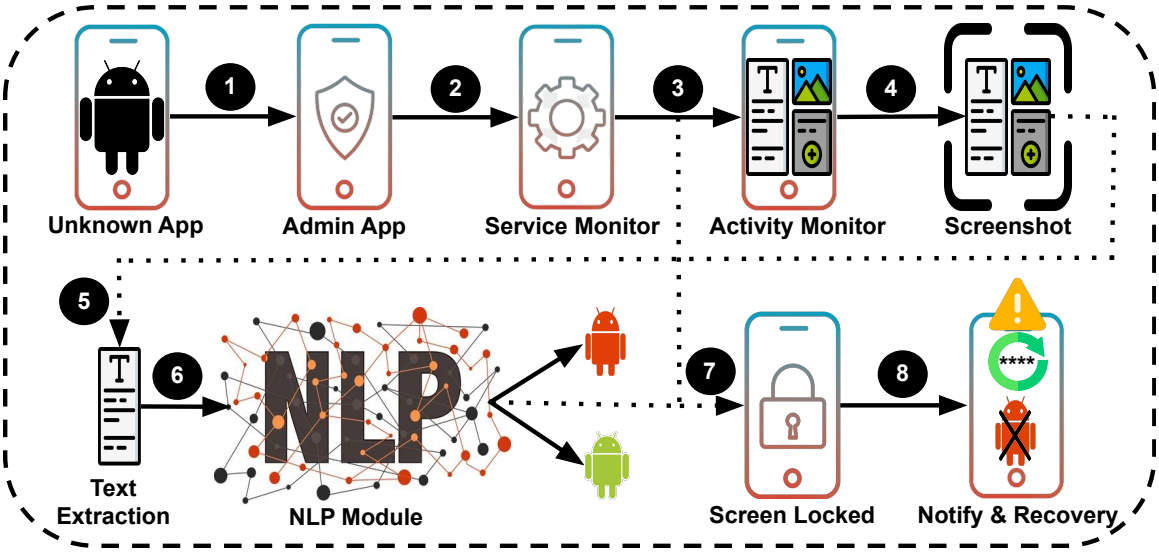


Figure 6.1: Overview of the sniper defense system.

The NLP unit harnesses transformers in order to accurately detect the appearance of ransom notes. An overview of our defense is shown in Figure 6.1.

6.3.1 Activity and Service Monitor

Our solution is designed to monitor newly installed apps and their associated actions that can result in a mobile device being locked. An important characteristic that distinguishes locker ransomware from other applications relates to its dependence on administrator privileges. This is because actions, such as locking a screen, setting password rules, and erasing data from a device often necessitate administrative access. As a result, step (1) of our detection process begins with tracking administrator privilege requests.

In Android, communication between running apps and the underlying OS is accomplished through intents. As such, our design monitors system broadcasts that involve the `DEVICE_ADMIN_ENABLED` intent as a way of tracking deployed apps and their associated privileges. Once a given app has received the `DEVICE_ADMIN_ENABLED` intent in response to an administrator request (`ACTION_ADD_DEVICE_ADMIN`), privilege is deemed granted, and the app is tagged for further analysis. At this point, our

design proceeds to the next step and monitors foreground service requests that are issued by the newly established administrator app. In addition to the aforementioned, our solution tracks pin code change requests. This is used to enable recovery and pin restoration in the event that the app is deemed malicious.

Unlike cryptographic ransomware that often runs as a background service, locker ransomware uses a different approach. It aggressively asserts itself as a prominent activity that overtakes the victim's screen as a mechanism for denying access to the compromised device. As such, foreground service requests represent another important feature for identifying locker ransomware behavior. Our design accounts for this activity under step ❷ of Figure 6.1.

In the event that an app elects itself to run as a foreground process, our design tags the app as suspicious and proceeds to step ❸. At this stage, our solution observes all of the activities the suspicious app displays to the user interface. To make our design efficient, our defense restricts its analysis to newly displayed activities and content updates that modify the device's user interface. To accomplish this, our design monitors the `TYPE_WINDOW_STATE_CHANGED` and `TYPE_WINDOW_CONTENT_CHANGED` events that are triggered when either a new window is launched or updated content is loaded. Hence, any time any of the aforementioned events are triggered, our solution captures the displayed content in the form of an image. It then extracts any text that may be embedded within the captured image and sends it to the NLP module for classification. These actions are illustrated in Figure 6.1 as steps ❹ – ❻.

6.3.2 Text Classification

The text classification subsystem serves the purpose of detecting ransom notes on an infected device. Once text has been extracted from the device's user interface, the data is consumed by an NLP unit for classification. To safeguard our design from text based attacks, we augment our system to be adversarially aware. As such,

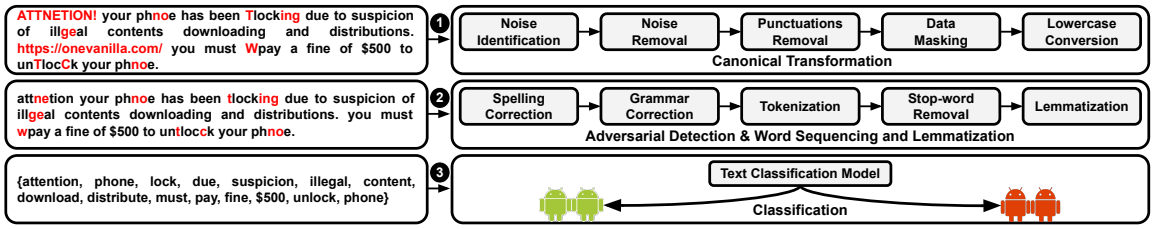


Figure 6.2: Main components of the natural language processing (NLP) module along with an example of how text from a ransom note is progressively transformed.

we harden our solution to be resilient against various adversarial machine learning (AML) attacks including character level attacks, word level attacks, and sentence level attacks. To support this approach, any text that is received by the NLP module undergoes three main phases: canoncial transformation, adversarial detection, and word sequencing with lemmatization. An overview of the main components that make up this module is shown in Figure 6.2.

Canonical Transformation. This phase entails transforming input data into a standard form that can be consumed by the classification model. It begins with the removal of noisy symbols that could impact the text classification process. For instance, HTML tags and other non-printable characters are removed from the original input. Similarly, symbols that are commonly used for punctuation, such as hyphens and accents, are eliminated from the text. Upon completion of the aforementioned pre-processing, the canonical transformation phase proceeds to masking parts of the data (data masking). This entails the removal of identifiers, such as uniform resource locators (URLs) and email addresses. The overall canonical transformation phase is concluded with a text standardization step. During this step, all of the retained text is made uniform by converting it to lowercase characters. This step simplifies the parsing of data in the later phases of the NLP module.

Adversarial Detection. One of our design objectives is to ensure sufficient protection against AML attacks. Research has shown that adding small perturbations to input are sufficient for an attacker to misclassify the input with high confidence

[123]. As such, an attacker can fool a detection system by carefully positioning perturbations in the form of characters and words into a ransom note. For instance, an attacker could leverage text from a previously released ransomware and replace a subset of its words with synonyms instead. Similarly, an attacker could harness different character manipulation techniques to evade detection without compromising the victim's ability to comprehend the ransom note [124]. This can be in the form of substitution, swap, insertion, and even deletion of characters in a given word.

To address the aforementioned concerns, our design takes into account the linguistic correctness of a given ransom note. As a result, any text that is received by this component is subjected to a series of rules that perform spelling and grammar corrections. To this end, our design discovers errors in text using a set of predefined rules referred to as bad rules. Such errors are then corrected using another set of predefined rules, namely, good rules. More specifically, our design accomplishes error detection and correction by breaking the input into chunks. Once split into chunks, each word in a given chunk is assigned a part-of-speech (POS) tag. The generated chunks are then compared against a rigorous set of error rules (bad rules). If a match occurs, an error is detected, and is subsequently corrected using the appropriate rule from the predefined set of good rules. This approach ensures that adversarially manipulated words through character swaps, deletions, and insertions are rectified prior to the text being used in the classification stage.

Word Sequencing and Lemmatization. This stage assumes the role of tokenizing the data received from the adversarial detection phase into a sequence of meaningful words [125]. The primary purpose of this step is to encode words, so that the context of a given phrase is preserved. The module also includes the removal of stop words (unimportant words), such as "the," "is," and "are" that don't add contextual meaning to a given sentence [126]. Stop word removal also has the benefit of improveing the latency of the overall system since this results in less words being processed by

the remaining stages of our design [127].

Once the stop words have been removed, the collected words (tokens) are transferred to the lemmatization step. This step assumes the role of mapping each word to its uninflected root (lemma). Since a given word could appear in multiple forms, it is essential to determine a common root that represents all the variations associated with a given word [128]. Therefore, lemmatization ensures that words, such as "gone" and "going" are mapped to a single root of the word, "go." This step also has the benefit of hardening the system against adversarial text attacks since this approach reduces the search space available to an adversary. Once this step has been completed, the lemmatized text is fed into a pre-trained NLP model for final classification as either benign or malicious.

6.3.3 Device Recovery and Pin Restoration

Our solution is designed to restore device access in the event that locker ransomware is detected. To this end, once the NLP module flags input text as being malicious, the entire app is consequently treated as malicious (step ❼). At this stage, further action is taken by our design to restore access to the compromised device (step ❽). Our recovery process involves a series of actions including permission revocation, app deletion, and user notification. For instance, if an app has been deemed malicious (e.g. created a malicious pop up that prevents device access), our design responds by immediately revoking the administrator privilege associated with the app. It then proceeds to deleting the app and all of its associated files. Once the app has been deleted, a system reboot is issued. This approach safeguards the user from accidentally paying the ransom. Once the device has been rebooted, the app is no longer considered active and an alert is sent to the user to inform them about the app removal.

Unlike the screen overlay approach that prevents the user from selecting other

apps, some ransomware families deny access to their victims by modifying the screen lock pin code of the device. To safeguard against this form of ransomware, our previously discussed service monitor assumes the role of tracking pin code change requests. As such, whenever a ransom note is detected and a pin code request was recorded, our solution resets the pin to a default value (e.g. 0000). To keep the user informed about this action, pin code resets are followed by a notification to the impacted user. The notification is used to inform the user about the newly established pin. On the other hand, if a given app is deemed benign in step ❼ and the pin code has not been changed, the previously captured image in step ❹ is discarded and a new evaluation cycle is initiated (transition from step ❼ back to step ❸). Our design continues to evaluate the running foreground app that has administrator privileges over a programmable duration (e.g. 30 seconds). If the app does not exhibit any malicious behavior over the aforementioned programmable period, the app is deemed benign.

6.4 Methodology

Training experiments were conducted on a system equipped with two Intel Xeon Gold 6152 processors, 768GB of main memory, and an NVIDIA Tesla V100 GPU. We re-purposed a diverse set of models that are popular in the field of natural language processing to perform ransomware detection. In our experiments, we re-purposed a convolutional neural network (CNN) [129], a standard recurrent neural network (RNN) [130], a long short term memory (LSTM) based RNN [131], and a gated recurrent unit (GRU) based RNN [132]. In addition to the aforementioned, we employed two transformer based models, namely BERT [75] and XLNet [76]. All of our models were pre-trained using the Google News word embedding [133] that corresponds to a 3 million-word corpus represented in the form of a 300-dimensional vector. We used TensorFlow 1.12.0 [134] with Python 3.6 for training the CNN and RNN based

Hyperparameter	RNN, LSTM, GRU	CNN	BERT, XLNet
# of epochs	100	50	5
Batch size	64	32	32
Embedding dim.	300	300	768
Optimizer	Adam	Adam	Adam
Learning rate	0.001	0.001	1e-5
Dropout rate	0.5	0.5	0.1

Table 6.1: Summary of hyperparameters used in all models.

models. BERT and XLNet, on the other hand, were trained using Pytorch 1.2.0 [135]. All of the models were trained with the Adaptive Moment Estimation optimizer (Adam). We explored different configurations including various batch sizes, learning rates, dropout rates, and epochs in order to identify the optimal settings for each model. To construct our dataset, we extracted ransom notes from 5,524 ransomware samples that were supplemented with 8,000 benign sentences from the Internet Movie Database (IMDB). We dedicated 80% of this dataset for training, 10% for validation, and 10% for testing. The final hyperparameters we deployed across the different models are summarized in Table 6.1.

We conducted experimental evaluations using 5,524 live samples that spanned 17 distinct ransomware families acquired from [117] and [60]. Our dataset consisted of both overlay screen and pin code ransomware. All of the ransomware samples we tested were executed on an Android 11 platform for a minimum of 10 minutes or until the display was locked. The Android image was rolled back to a pristine snapshot after every executed sample. This was performed to reduce the possibility of previously executed ransomware interfering with subsequent runs. The ransomware families we used and their corresponding characteristics are listed in Table 6.2. In addition to ransomware samples, we tested 220 benign apps in order to assess the accuracy of our defense in terms of false positives. More specifically, we evaluated applications from multiple categories including health and fitness, productivity, social, entertainment, travel and local, and communication. The aforementioned categories

Family	Samples	Lock Screen		Payment Method		Threat Msg	Defeated
		Overlay Screen	Change Pin	Trad.	Elect.		
Aples	33	✓	✗	✗	✓	✓	✓
Charger	11	✓	✗	✗	✓	✓	✓
DoubleLocker	16	✗	✓	✗	✓	✓	✓
Fusob	1,354	✓	✗	✗	✓	✓	✓
Jisut	777	✓	✗	✓	✓	✓	✓
Koler	483	✓	✗	✗	✓	✓	✓
LockDroid	215	✓	✗	✗	✓	✓	✓
LockerPin	54	✗	✓	✓	✗	✓	✓
MalLocker	7	✓	✗	✗	✓	✓	✓
Pletor	34	✓	✗	✗	✓	✓	✓
PornDroid	241	✓	✗	✗	✓	✓	✓
SimpleLocker	946	✓	✗	✓	✗	✓	✓
Slocker	699	✓	✗	✓	✗	✓	✓
Svpeng	22	✓	✗	✓	✗	✓	✓
WannaLocker	14	✓	✗	✗	✓	✓	✓
WipeLocker	7	✓	✗	✗	✓	✓	✓
Xbot	13	✓	✗	✗	✓	✓	✓
Other	598	N/A	N/A	N/A	N/A	N/A	N/A
Total Samples	**5,524**						

Table 6.2: Summary of locker ransomware families and their capabilities.

and their corresponding applications are summarized in Table 6.3.

We implemented a prototype of our solution on a real platform. We deployed our solution on a Stratus C5 Elite smartphone that ran Android 11. We used the Tesseract-OCR 5.0.0 tool [136] to extract text displayed on the device's user interface. Since our models were trained to process text written in the English language, we leveraged the GoogleTrans 3.0.0 tool [137]. This allowed us to detect input text written in other languages and translate it into English prior to running it through the NLP model. Furthermore, we characterized the runtime and energy overhead of different NLP models in order to identify their suitability for deployment on mobile devices. We ran a diverse set of test suites available in PassMark [120], a commonly used benchmark for measuring the performance of Android devices. This benchmark is equipped with a comprehensive set of CPU, memory, I/O, and graphics workloads.

Category	Applications
Social	*Facebook, Twitter, Linkedin, Pinterist, TikTok, Instagram, Reddit, Bigo Live, Tumblr, Bumble, Uhive, Vero, Hootsuite, Tinder, Playsee, Lokaleo, Scenes*
Communication	*WhatsApp, Messenger, Snapchat, Ring, Telegram, Discord, Viber, Google Meet, Google Chat, Google Voice, TextNow, GroupMe, Speakeasey, MyMoov, KakaoTalk, Truecaller, Microsoft Teams, Skype, WeChat, CallApp, Line, Zoom, Slack*
Productivity	*Gmail, Dropbox, Calendar, Todoist, Google Translate, Easy Scanner, Adobe Acrobat, QR Scanner, Google Docs, Google Chrome, Microsoft Authenticator, Find My Device, Outlook, PaperColor, Google Tasks, Wix Owner, Picsart, Word, Excel, PowerPoint, Forest,PDF, Google Slides, Deezer, TMZ, TickTick, Time Planner, WPS Office,VLC, Sublime, Evernote, WordPress, CapCut, Audiomack, SoundCloud, Paintful, Canva, YouCut, MX Player, Rumble, StarMaker, StayFree, Pocket FM, Baya, Scribd,Audible*
Entertainment	*YouTube, Wordscapes, HBO, Twitch, Tubi, Pluto TV, JustWatch, Amazon Prime Video, Bowmasters, Toon Blast, Car Stunts, Showtime, Starz, Subway Surfers, Candy Crush, Movies Anywhere, SiriusXM, Boomerang, Magic Tiles, YouTube Kids, My Talking Tom, Snow Race, Triple Tile, Crunchyroll, Pandora, DisneyNOW, Globoplay, Vix, SLING, Google TV, Xbox, Vudu, IMDb, Shudder, Angry Birds, Temple Run, Traffic Racer, Thief Puzzle, Spotify, Bubble Shooter, Plex*
Travel & Local	*Google Maps, TripAdvisor, Uber, Yelp, Expedia, Lyft, Booking, trivago, BlaBlaCar, Triplt, Kayak, Wyndham, FlixBus, Airbnb, CheapFlights, Skyscanner*
Finance	*Cash App, Vinmo, PayPal, Zelle, Chime, Google Pay, Capital One, Chase, Experian, Credit Karma, Varo Bank, Mint, Google Wallet, Bank of Amarica, Rocket Money,Cleo, NerdWallet, Webull, Earnin, Empower, Credit Sesame, UNest, Stach, Albert, Kolver*
Shopping	*Amazon, Wallmart, SHEIN, Target, Offerup, Alibaba, Shopify, Aliexpress, Poshmark, Nike, Home Depot, Five Below, eBay, Shop, Sam's Club, Fashion Nova, Wish, Mercari*
Food & Drink	*McDonald's, Foodpanda, Uber Eats, Domino's Pizza, Subway, Dunkin, Whisk, KFC, Tasty, Starbucks, Dairy Queen, OpenTable, DoorDash, Burger King, HelloFresh*
Education	*Duolingo, Google Classroom, Toca Life World, Rosetta Stone, StudySmarter, Elevate, Sololearn, Coursera, ABC Kids, Babbel, Quora*
Health & Fitness	*Google Fit, FitnessPal, Blood Pressure, Sweatcoin, FreshTri, Peloton, Tracker, Lifesum*

Table 6.3: Summary of benign mobile applications and their corresponding categories.

As such, this benchmark allowed us to systematically characterize our design's sensitivity to different workload types. The aforementioned test suites are listed in Table 6.4.

Suite	Benchmark
CPU	Integer Math (IM), Floating Point Math (FPM), Find Prime Numbers (FPNs), Random String Sorting (RSS), Data Encryption (DE), Data Compression (DC), Physics, Extended Instructions (EIs), Single Thread (ST)
Memory	Database Operations (DOs), Memory Read Cached (MRC), Memory Read Uncached (MRU), Memory Write (MW), Memory Latency (ML), Memory Threaded (MT),
I/O	Internal Storage Read (ISR), Internal Storage Write (ISW), External Storage Read (ESR), External Storage Write (ESW)
2D Graphics	Complex Vectors (CVs), Transparent Vectors (TVs), Solid Vectors (SVs), Image Rendering (IR), Image Filters (IFs)
3D Graphics	Simple Test (ST), Complex Test (CT), OpenGL

Table 6.4: Summary of performance benchmarks.

6.5 Evaluation

6.5.1 Model Robustness

Non-deep Learning Models. Although our study focused on the use of deep learning models, we initially explored lightweight classifiers, such as random forest, support vector machine, and logistic regression to understand their effectiveness in detecting locker ransomware. The results of this experiment along with the models we tested are shown in Figure 6.3. Overall, we observed that the models in the aforementioned Figure performed well when tested against a non-adversarial test set (*Non-Adv*). This test resulted in detection rates that ranged between 85.2% and 97.2% with a geometric mean of 95.0%. Although lightweight models are attractive for deployment in resource constrained mobile devices, it is important to understand their robustness against adversarial machine learning attacks. To this end, we augmented our non-adversarial test set (*Non-Adv*) with different types of perturbations to produce three AML test sets: DeepWordBug (*DWB*) [138], Genetic Algorithm (*GA*) [139], and Probability Weighted Word Saliency (*PWWS*) [140].

Our first AML test set, *DWB*, is a test set that focuses on character level text

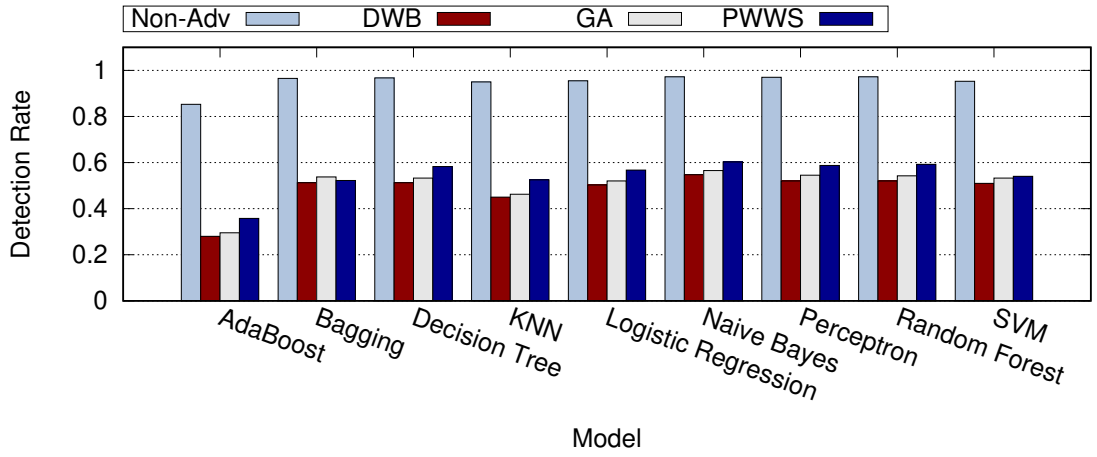


Figure 6.3: Summary of detection rates for non-deep learning models with adversarial and non-adversarial test sets.

attacks. It evades detection by manipulating key words in a given text through two phases. The first phase entails identifying the importance of words in a given text based on equation (6.1). In this equation, the importance of the i-th word in a given text $(x_1, ..., x_n)$ is computed using the $Score(x_i)$. λ is a hyperparameter, and F is the model's function that maps an input X to a label Y. An example of adversarial text generated using this approach is shown in Figure 6.4.

$$Score(x_i) = [F(x_1, \ldots, x_i) - F(x1, \ldots, x_{i-1})] + \lambda[F(x_i, \ldots, x_n) - F(x_{i+1}, \ldots, x_n)] \quad (6.1)$$

Overall, our results show that the non-deep learning models listed in Figure 6.3 were less effective in detecting adversarially designed inputs. For instance, the aforementioned models yielded detection rates that ranged between 28.0% and 54.8% when evaluated against the *DWB* test set. In general, the detection rates did not exceed 54.8%. This correlates to more than a 40% drop relative to what we observed with the non-adversarial test set (*Non-Adv*). In our experiments, we found that one of the simplest models, naïve bayes, exhibited the best performance which yielded a mere

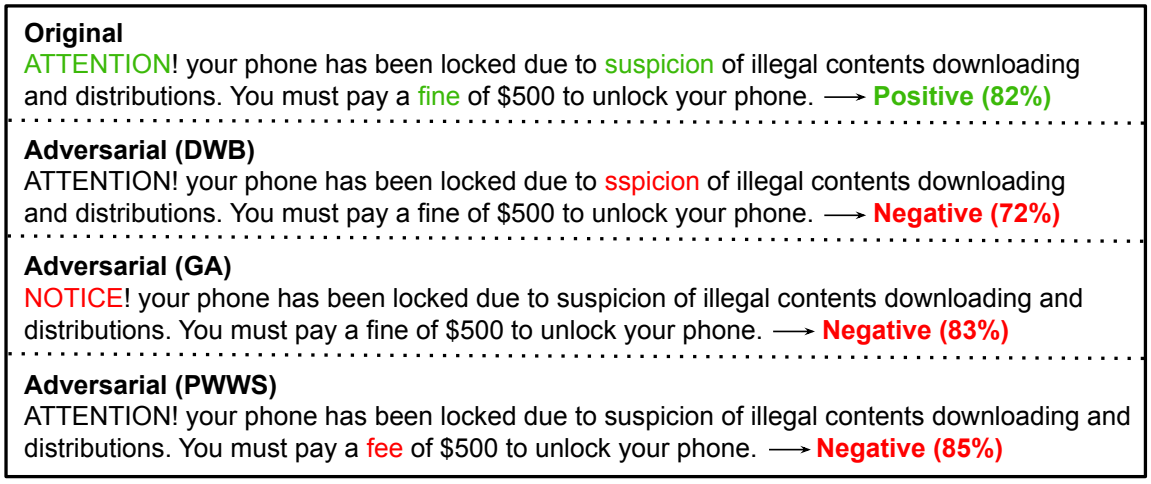


Figure 6.4: Original text example and the corresponding adversarial samples generated using DWB, GA, and PWWS.

detection rate of 54.8%. Popular algorithms, such as SVM performed even more poorly, since it was incapable of detecting no more than 51.0% of the adversarially designed malicious samples.

In addition to *DWB*, we explored the use of a genetic algorithm (*GA*) to produce supplemental adversarial samples. More specifically, we used the nearest neighbor replacement algorithm defined in equation (6.2) for generating our *GA* based test set. In this equation, X_{adv} represents the ideal adversarial sample that would deceive a target model f into giving false results. $P^{(g-1)}$, on the other hand, denotes the $(g-1)^{th}$ generation. Unlike *DWB*, *GA* is a word level attack that makes use of the Euclidean distance to reduce the difference in semantics between the original word and its adversarial replacement. An example of adversarial text generated using the *GA* approach is shown in Figure 6.4.

$$X_{adv} = P^{(g-1)}_{\arg\max f(P^{g-1})_{target}} \tag{6.2}$$

Similar to the *DWB* experiment, our *GA* results yielded detection rates that ranged between 29.5% and 56.5%. On average, we observed a detection rate of 50.4% across the different models. Analogous to the *DWB* experiment, naïve bayes offered

the best performance (56.5%) while adaptive boosting performed the worst (29.5%). Analogous to character level attacks, our *GA* results demonstrate that attackers can adversarially compromise non-deep learning models through word level attacks.

Probability Weighted Word Saliency (*PWWS*) is another approach that our study considered for producing adversarial data. Similar to *GA*, *PWWS* is a word-level attack that evades detection by replacing words with their synonyms using WordNet 2. This approach also relies on an optimization procedure for finding the appropriate substitutions by maximizing the word saliency from a set of selected synonyms. Although *PWWS* is a word-level attack, we extend this method to also perform sentence level attacks by perturbing the ordering of the words in a given phrase.

$$R(w_i, L_i) = \arg\max\ P(Y_{true}|x) - P(Y_{true}|x'_i) \tag{6.3}$$

The optimization procedure used in this method is illustrated in equation 6.3. In this equation, $R(w_i, L_i)$ represents the best replacement synonym w_i^* of the i^{th} word in a given text x. x'_i is obtained by replacing the i^{th} word in x with each candidate synonym. $P(Y|x)$ is the probability for classifying text x, such that the target model can be fooled. An example of adversarial text generated through *PWWS* is shown in Figure 6.4.

We observed that the non-deep learning models shown in Figure 6.3 were slightly more resilient in detecting adversarial attacks, but still performed poorly. The aforementioned models yielded detection rates that ranged between 35.8% and 60.3% when evaluated against the *PWWS* test set. On average, the models showed a 3.8% and 5.8% improvement in detection rates relative to *DWB* and *GA*, respectively. However, despite this slight improvement, our results show that non-deep learning models are still vulnerable to adversarial samples generated using the *PWWS* approach. Overall, our results underscore the need for superior models that can offer better protection against AML generated data sets.

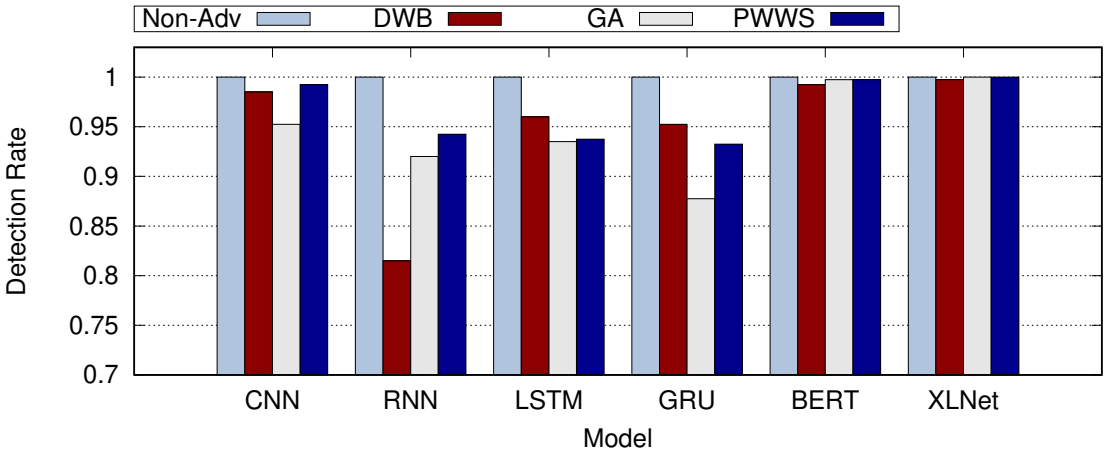


Figure 6.5: Summary of detection rates for deep learning models with adversarial and non-adversarial test sets.

Deep Learning Models. We examined a diverse set of deep learning models that were trained to detect locker ransomware. We evaluated a convolutional neural network (CNN) [129], a standard recurrent neural network (RNN) [130], a long short term memory (LSTM) RNN [131], and a gated recurrent unit (GRU) RNN [132]. In addition to the aforementioned, we employed two transformer based models, BERT [75] and XLNet [76]. All of the previously mentioned models were evaluated against commonly used quality metrics that include: accuracy, true positive rate (TPR), false positive rate (FPR), precision, recall, F-score, and area under the curve (AUC). Furthermore, all of the aforementioned models were tested against the same non-adversarial and adversarial datasets: non-adversarial (*Non-Adv*), DeepWordBug (*DWB*), Genetic Algorithm (*GA*), and Probability Weighted Word Saliency (*PWWS*).

Our results indicate that all of the deep learning models achieved ideal performance under the *Non-Adv* dataset (100% detection rate). Unlike the non-parametric models, none of the deep learning designs yielded any false positives or negatives while using the non-adversarial dataset. This correlates to a 5% improvement relative to the rates achieved by the non-parametric models previously depicted in Figure 6.3. In addition, to the *Non-Adv* dataset, we observed a significant improvement in the

ability to detect adversarially designed ransom notes. On average, the deep learning models demonstrated a 45% increase in the ability to detect malicious samples that span the *DWB*, *GA*, and *PWWS* datasets.

Figure 6.5 reveals that XLNet outperformed the remaining models with BERT being a close second. Our results demostrate that the transformer based models offered better performance over their CNN and RNN counterparts across all the metrics depicted in Table 6.5. For instance, XLNet and BERT detected 99.7% and 99.2% of the *DWB* samples. This correlates to XLNet and BERT having one and three false negatives, respectively. The CNN model, on the other hand, experienced a slightly lower detection rate (98.5%) which corresponds to six malicious samples going undetected. Unlike the transformer and CNN models, however, combining our solution with RNNs resulted in suboptimal detection rates. Although such models previously represented the state-of-the-art in the field of natural language processing, the standard, LSTM, and GRU based RNNs misclassified 74, 16, and 19 *DWB* samples, respectively. This data suggests that RNNs are less effective in combating sophisticated forms of locker ransomware that employ AML techniques to evade detection.

A similar trend was observed with the *GA* and *PWWS* datasets. Overall, our transformer based models performed slightly better while using the aforementioned datasets. We observed that XLNet classified all of the *GA* and *PWWS* samples correctly. Similarly, BERT exhibited near ideal detection rates by misclassifying only one malicious sample from each dataset. The CNN model, on the other hand, experienced a 3% drop while using the *GA* dataset (95.2%). The CNN model, however, improved its detection rate to 99.2% while using the *PWWS* dataset. This correlates to only three samples being undetected. On the other hand, the RNN models manifested suboptimal detection rates that ranged between 87.7% and 94.2%. This correlates to an average that is slightly above 92% when evaluated with samples from the *GA* and *PWWS* datasets.

Overall, our results suggest that XLNet demonstrated the most resilience against adversarial attacks. Although the BERT and CNN models produced competitive detection rates, XLNet consistently outperformed the aforementioned models across a diverse set of evasion techniques that span, character (DWB), word (GA), and sentence ($PWWS$) level attacks. Although devising malware defenses that are robust to adversarial machine learning attacks is critical, other factors, such as energy and runtime must be considered as part of proposing a solution that is suitable for mobile platforms.

Model Runtime and Energy. The performance overhead of our defense system is primarily a function of the runtime associated with the text classification module. To this end, we conducted runtime and energy efficiency experiments across the different NLP models in order to understand their suitability for deployment on mobile devices. Figure 6.6 shows the runtime of different NLP models based on measurements from our prototype system.

Overall, we observed that XLNet achieved the best performance in terms of runtime. On average, XLNet only required 21 ms to run text classification tasks to completion. BERT, on the other hand, suffered an additional 5% overhead while attempting to classify the same tasks. This overhead increased significantly while using the CNN and RNN models. For instance, we observed that the CNN model required almost 27 ms to run to completion. This correlates to 27% and 21% increases in runtime overhead relative to the transformer based models, XLNet and BERT. In general, the RNN models proved to be the least efficient, offering the worst execution times. For example, the GRU model took 33 ms to classify text, on average. This corresponds to a 56% increase relative to what we observed with XLNet. Similarly, the LSTM and standard RNN models experienced comparable overheads. We observed 33 ms and 31 ms for the LSTM and standard RNN models, respectively. This is not surprising given that a typical RNN neuron by design requires 8x the number

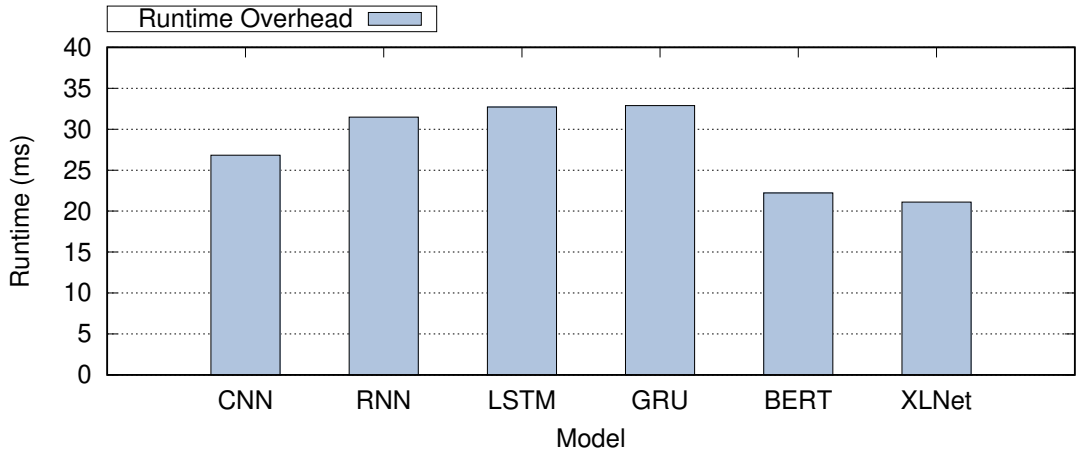


Figure 6.6: Summary of the runtime for different NLP models.

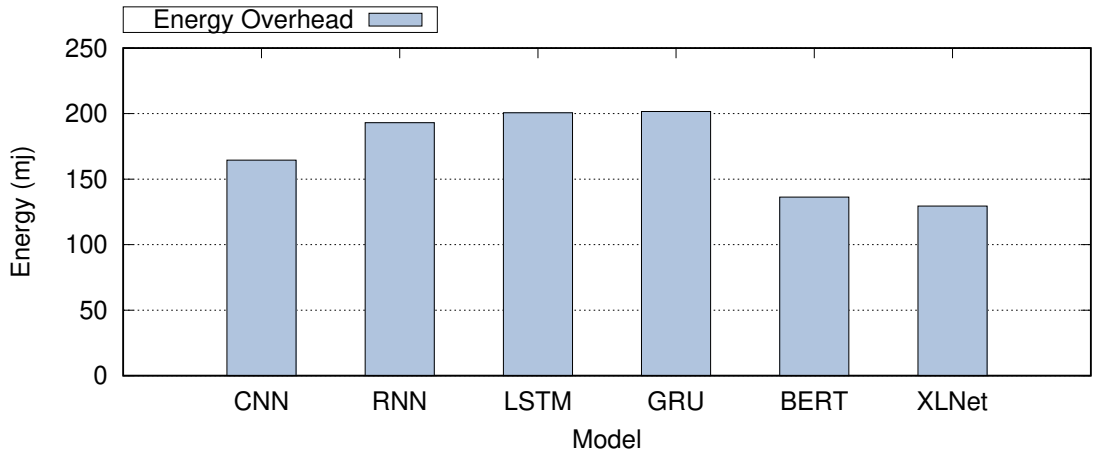


Figure 6.7: Summary of the energy for different NLP models.

of weights and multiply-accumulate operations of a standard CNN cell [141].

Figure 6.7 shows the energy consumption of the previously discussed NLP models. Overall, our energy results correlate to our runtime findings. We observed that the transformer based models offered the best energy efficiency while the RNN models offered the worst. To put things in perspective, both XLNet and BERT consumed 129 mj and 136 mj, respectively. This overhead increased to 164 mj while using a CNN. However, significantly more energy was consumed while using a GRU. For instance, the GRU expended 20% more energy (202 mj) compared to its CNN counterpart. Although the LSTM and standard RNN models consumed slightly less energy, they

still required 201 mj and 193 mj, respectively. Overall, we found that in addition to robustness to AML attacks, XLNet is the most energy efficient NLP model. This makes XLNet the most suitable choice for deployment on mobile systems that are often resource and energy constrained.

Our defense has three main sources of runtime and energy overhead. These sources correspond to performing canonical transformation, adversarial detection, and text classification. Figures 6.8 and 6.9 show a breakdown of the runtime and energy overhead for our design. The first source of runtime and energy overhead relates to the canonical transformation phase. This phase is responsible for transforming input data into a standard form that can be consumed by the classification model. We measured runtime and energy costs of 38 ms and 233 mj. The second source of overhead relates to the adversarial detection phase. This phase involves performing spelling and grammar correctness to ensure linguistically sound text data. This phase marks the biggest source of overhead within our design. We observed runtime and energy costs of 72 ms and 443 mj. This is 1.9x the runtime and energy consumption of the canonical transformation phase. However, despite the high overhead, this phase is critical to our design since it represents the core component for defending against AML attacks. The final source of overhead within our design relates to classification. On average, this phase required 21 ms and 129 mj to classify 10K sentences from a mix of ransomware and benign applications while using the XLNet model. Overall, our end-to-end design required 131 ms and 805 mj to perform complete detection of ransomware samples. It is important to note that in most cases, a given user attempting to launch an app will not result in any of these phases being executed. The aforementioned overhead is introduced only when a launched application is deemed suspicious.

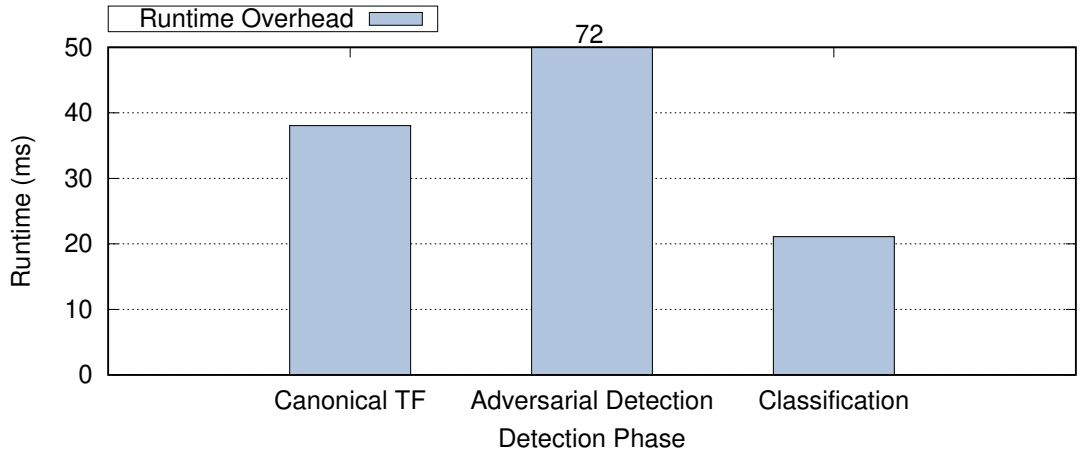


Figure 6.8: Breakdown of the runtime overhead for our NLP module phases.

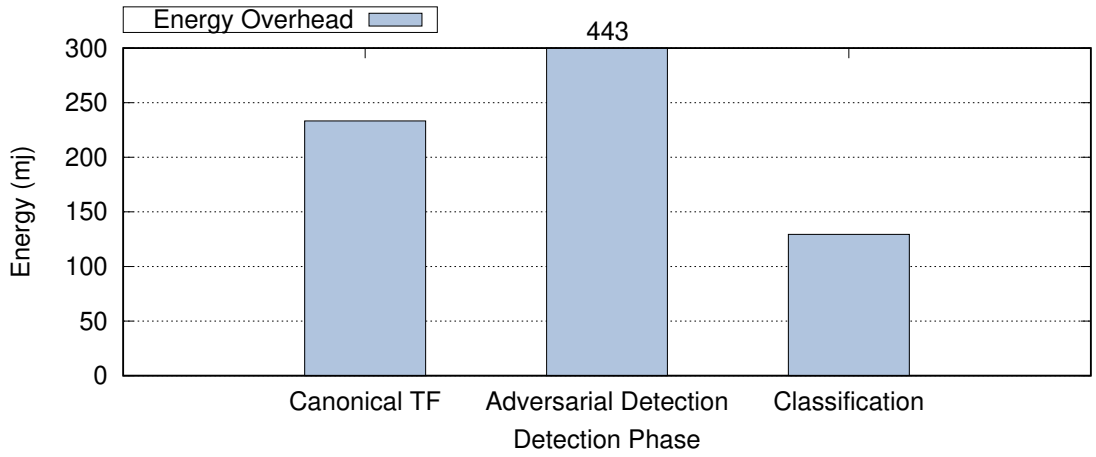


Figure 6.9: Breakdown of the energy overhead for our NLP module phases.

6.5.2 Workload Characterization

Computational Workloads. We ran a wide range of experiments that aimed to evaluate the performance impact of our system on CPU workloads. The CPU workloads we used are listed in Table 6.4. Figure 6.10 summarizes the performance overhead of our solution on CPU workloads relative to an unsecure baseline. In general, most of the CPU-bound workloads incurred an insignificant amount of overhead when dispatched on our solution. We found that our system incurred a negligible overhead that was well under 2% across most programs with the exception of `Integer Math (IM)` and `Data Encryption (DE)`. `Integer Math (IM)` is a workload that aims to

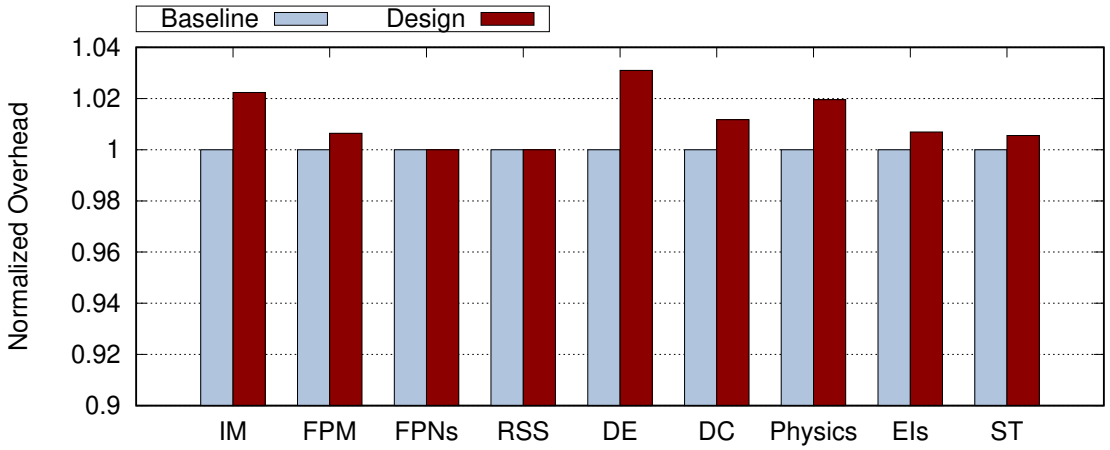


Figure 6.10: Performance overhead on CPU workloads relative to an unsecure baseline.

evaluate how fast a CPU can perform arithmetic operations, such as addition, multiplication, and division. `Data Encryption (DE)`, on the other hand, is a workload that encrypts and hashes blocks of random data using algorithms, such as AES, ECDSA, and SHA256. Unlike the other programs, the `Integer Math (IM)` and `Data Encryption (DE)` workloads incurred 2.2% and 3.0% reduction in performance, respectively. We attribute this reduction to our design continuously running in the background in order to monitor service requests issued by a user's apps.

Memory Workloads. We also ran various workloads that aimed to systematically evaluate the impact of our solution on the memory subsystem. The memory workloads we used for this purpose are listed in Table 6.4. Figure 6.11 summarizes the performance overhead of our solution on memory centric workloads relative to an unsecure baseline. On average, our solution exhibited a performance impact that was below 3.2%. The majority of the workloads encountered overheads that were well below 3% with the exception of the `Database Operations (DOs)` workload, which experienced a 9.0% overhead. We attribute this reduction in performance to the size of our NLP module. Since our design relies on a pre-trained model, a substantial amount of memory is needed in order to fit all of the hyperparameters in memory.

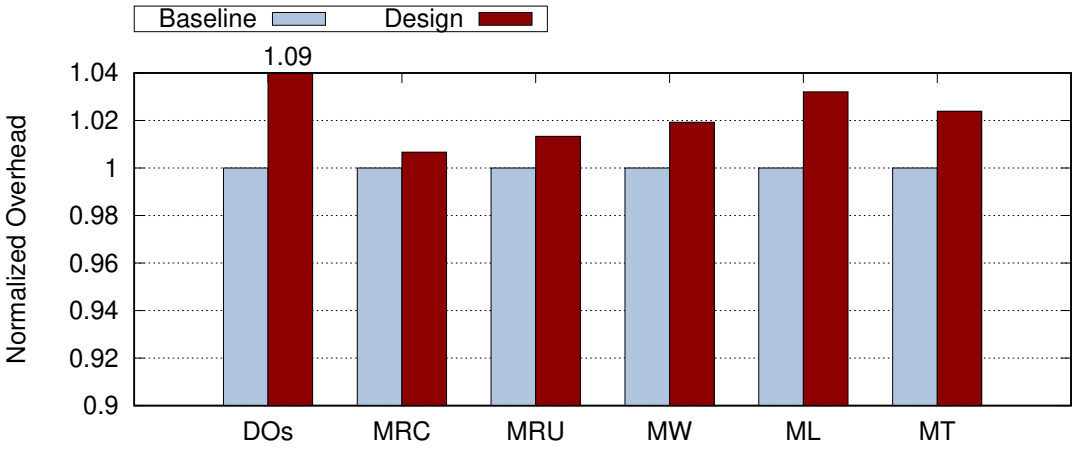


Figure 6.11: Performance overhead on memory workloads relative to an unsecure baseline.

This can induce low memory conditions when combined with memory intensive apps, such as database applications.

I/O Workloads. We also conducted tests that assessed the performance impact of our design on I/O. The I/O workloads we used can also be found in Table 6.4. Figure 6.12 summarizes the performance overhead of our solution on the I/O subsystem relative to an unsecured baseline. We observed an average performance decline that was less than 2.8%. The majority of workloads experienced overhead that was well below 3%, with the exception of `External Storage Read (ESR)` and `External Storage Write (ESW)`, which had performance losses of 4.8% and 3.3%, respectively. This overhead is introduced as a result of our continuous evaluation of newly installed apps for suspicious behavior.

Overall Performance Impact. Figure 6.13 illustrates the overall performance impact of our design relative to an unsecured baseline for different subsystems. In general, our design had the least impact on the CPU subsystem with an average performance reduction that is under 1.2%. Additionally, we observed that our design exhibited a 2.8% loss in performance across I/O workloads. Similarly, our design experienced slightly elevated overheads while running memory bound workloads, av-

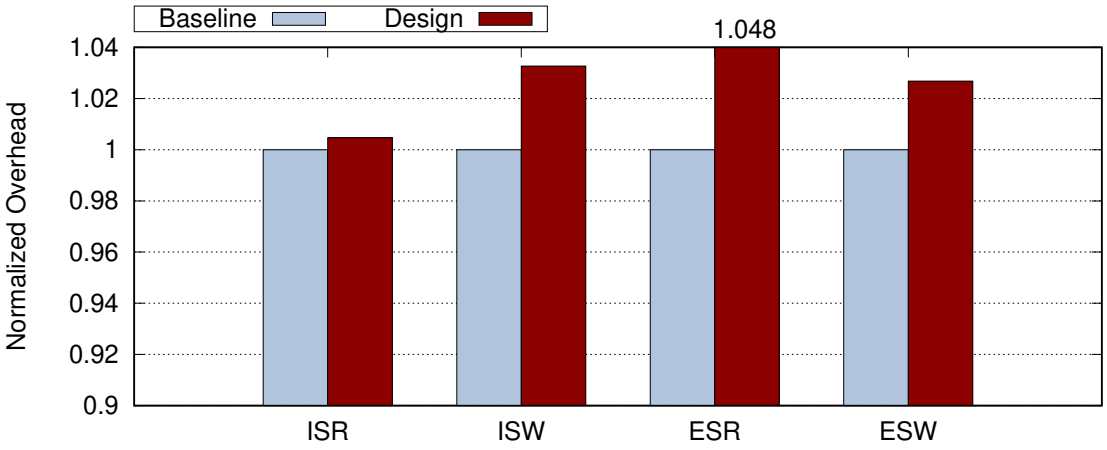


Figure 6.12: Performance overhead on I/O workloads relative to an unsecured baseline.

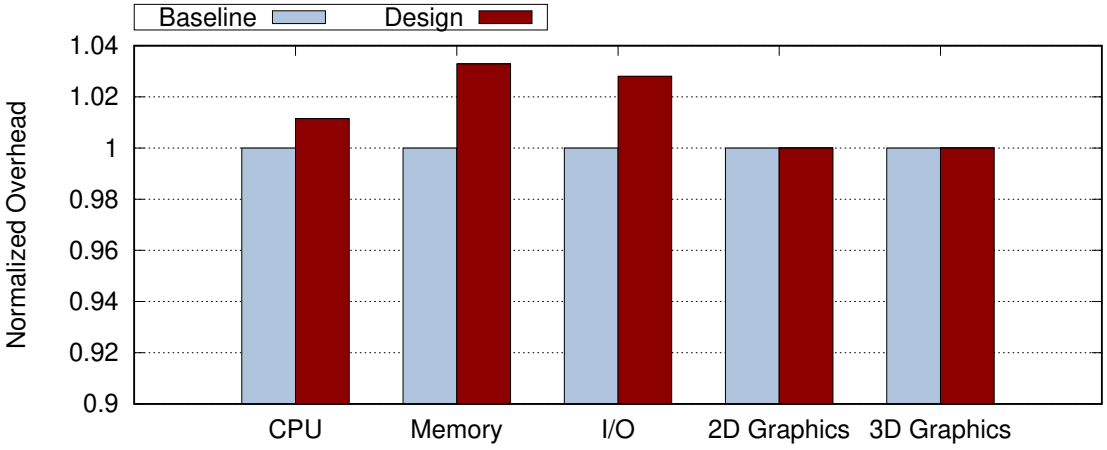


Figure 6.13: Overall performance overhead relative to an unsecured baseline.

eraging a 3.2% drop in performance. On the other hand, we observed that our design had no impact on the performance of 2D and 3D graphic workloads. This is expected since our design does not rely on the device's GPU for any of its analysis. Overall, our design experienced a reduction in performance that was less than 1% when considering all of the subsystems together. Such negligible overheads demonstrate the efficiency of our proposed solution.

Model	Accuracy	TPR	FPR	Precision	Recall	F-score	AUC
CNN (Non-Adv)	1	1	0	1	1	1	1
CNN (DWB)	0.997	0.985	0	1	0.985	0.992	0.992
CNN (GA)	0.990	0.952	0	1	0.952	0.975	0.976
CNN (PWWS)	0.998	0.992	0	1	0.992	0.996	0.996
RNN (Non-Adv)	1	1	0	1	1	1	1
RNN (DWB)	0.963	0.815	0	1	0.815	0.898	0.907
RNN (GA)	0.984	0.920	0	1	0.920	0.958	0.960
RNN (PWWS)	0.988	0.942	0	1	0.942	0.970	0.971
LSTM (Non-Adv)	1	1	0	1	1	1	1
LSTM (DWB)	0.992	0.960	0	1	0.960	0.979	0.980
LSTM (GA)	0.987	0.935	0	1	0.935	0.966	0.967
LSTM (PWWS)	0.987	0.937	0	1	0.937	0.967	0.968
GRU (Non-Adv)	1	1	0	1	1	1	1
GRU (DWB)	0.990	0.952	0	1	0.952	0.975	0.976
GRU (GA)	0.975	0.877	0	1	0.877	0.934	0.938
GRU (PWWS)	0.986	0.932	0	1	0.932	0.965	0.966
BERT (Non-Adv)	1	1	0	1	1	1	1
BERT (DWB)	0.998	0.992	0	1	0.992	0.996	0.996
BERT (GA)	0.999	0.997	0	1	0.997	0.998	0.998
BERT (PWWS)	0.999	0.997	0	1	0.997	0.998	0.998
XLNet (Non-Adv)	1	1	0	1	1	1	1
XLNet (DWB)	0.999	0.997	0	1	0.997	0.998	0.998
XLNet (GA)	1	1	0	1	1	1	1
XLNet (PWWS)	1	1	0	1	1	1	1

Table 6.5: Summary of quality metrics of NLP models using Non-Adv and DWB, GA and PWWS adversarial test sets.

6.5.3 Discussion

Our study entailed conducting a comprehensive set of experiments designed to evaluate the ability of our solution in detecting and recovering from locker ransomware. Our validation included over 200 benign applications and more than 5,000 ransomware samples that spanned 17 ransomware families. The different ransomware families we used are summarized in Table 6.2.

Altogether, our results demonstrate that our design is reliable in differentiating between benign and ransomware activity. Our solution accurately classified all of the malicious samples that we ran, including overlay screen and change pin-code ran-

Evaluation	*Results*
Total Samples	5,744
Ransomware Samples	5,524
Benign Applications	220
False Positives	0.0%
False Negatives	0.0%

Table 6.6: Summary of detection results.

somware. Similarly, our design demonstrated resiliency with regards to misclassifying benign apps. Our system was able to correctly classify all of the benign applications without triggering any false positives. During this process, we observed that less than 1% of the benign applications we tested requested administrator privileges. This correlates to only two apps out of a total of 220 applications that requested such privileges. More specifically, Microsoft Authenticator, a multifactor authentication app that enables users to securely sign into their accounts was one of the applications that requested administrator privileges. This app requests administrator privileges in order to collect GPS data and enable location based authentication. The second app that required administrator privileges was Find My Device. Since this is an app that allows users to locate their lost devices and lock them until they are found, administrator privileges are requested. Our solution was also able to correctly classify such apps without yielding any false positives. The results of the entire experiment are summarized in Table 6.6.

In addition, our solution was able to recover from all of the ransomware samples we tested, including overlay screen and change pin-code ransomware. We observed that all of the ransomware samples we encountered requested administrator privileges upon installation, in order to control the screen lock action. In the case of overlay screen ransomware, all of the samples leveraged the foreground service for displaying their ransom notes to the user interface. For instance, when we tested Svpeng [121], a ransomware that infiltrates Android smartphones through a fake Adobe Flash update

message, it locked the UI display and popped up a bogus FBI message that demanded a $200 ransom. In addition to Svpeng, our system was able to detect malicious activity from all samples within 131 ms of the screen being locked. In all cases, our solution successfully restored access to the infected system within the aforementioned period by revoking the administrator privileges, deleting the offending app along with its associated files, and issuing a notification to the user.

We also observed that some ransomware samples changed the pin code in order to prevent users from accessing their devices. For example, we tested Lockerpin [122], a ransomware that changes the pin of an infected device followed by a $500 ransom demand. Our design was able to detect such activity within 131 ms of the screen being locked. It removed the malicious app and all of its related files after revoking administrator privileges, reset the screen lock pin code with a new pin code of 0000, and notified the user about the app removal and the newly established pin.

CHAPTER VII

Conclusion and Future Work

In this work, we propose several novel runtime solutions that autonomously defend against various ransomware attack variants, including cryptographic, corruption, and locker ransomware attacks. Unlike prior work that can leave victims with partially impacted filesystems or locked devices, our proposed solutions seamlessly preserve the compromised system without having to undergo an explicit recovery process or manual user intervention. We leverage the use of behavioral analysis and artificial intelligence techniques for defending against different types of ransomware attacks, including cryptographic, corruption, and locker ransomware attacks. We extensively validate the robustness of the proposed solutions against thousands of recently released ransomware samples in the wild and show that our solutions reliably recover from all the malicious samples that we tested while tolerating malicious evasion techniques. We demonstrate the robustness of the proposed solutions against misclassifying benign applications as ransomware. We also evaluate the performance overhead of the proposed solutions against a diverse set of standard benchmarks and demonstrate that our proof-of-concept implementation incurs negligible overhead while running a diverse set of realistic workloads commonly used for measuring performance.

Defending against malware attacks is an ongoing challenge in the field of cybersecurity, as attackers continually evolve their tactics and techniques. While significant

progress has been made in developing defensive measures, there are several avenues for future work that can further enhance our ability to protect systems and networks from malware threats. In particular, we highlight the following potential avenues for future research and development in the field of defending against malware attacks.

Advanced Machine Learning Techniques. Although machine learning has proven to be effective in mitigating malware based on patterns and behaviors. Future research will concentrate on developing more advanced machine-learning algorithms and models that can effectively mitigate emerging malware attacks. This includes exploring deep learning techniques, reinforcement learning, and ensemble methods to mitigate emerging malware threats. Additionally, future research will concentrate on the development of lightweight and resource-efficient machine-learning models that can be deployed on resource-constrained devices.

Zero-Day Malware Mitigations. As malware threats constantly evolve, future work will focus on the development of proactive defense mechanisms that can effectively detect and mitigate zero-day malware attacks. This involves the use of behavior-based analysis, anomaly detection, and heuristic techniques to identify suspicious activities and uncover unknown malware threats.

Adaptive and Resilient Defenses. As malware attacks constantly evolve, future research will concentrate on developing intelligent defense mechanisms that can dynamically adapt to changing malware landscapes. This includes exploring techniques such as dynamic sandboxing, virtual patching, machine learning-based anomaly detection, behavior-based analysis, and real-time response systems that can detect and mitigate malware attacks in real-time.

Malware Defenses for Emerging Technologies. As new technologies emerge, such as artificial intelligence, Intelligent vehicles, and the Internet of Things (IoT), defending against malware in these domains presents unique challenges including resource constraints, diverse architectures, and communication protocols. Future work

will focus on understanding the specific vulnerabilities and attack vectors associated with these technologies and developing specialized defense mechanisms. This includes securing smart devices, autonomous vehicles, and AI-driven systems against emerging malware attacks.

Cloud-Based Malware Defenses. As cloud computing continues to grow, defending against malware in cloud environments becomes increasingly important. Future work will address the unique challenges of cloud-based malware defense, including securing virtualized environments, detecting lateral movement within the cloud, and protecting shared resources. Research efforts will focus on developing scalable and efficient cloud security solutions, including intrusion detection and prevention systems, behavior-based anomaly detection, and real-time monitoring mechanisms.

Privacy-Preserving Malware Defenses. Malware defense systems often require access to sensitive data, which raises concerns about user privacy. Future work will emphasize the development of privacy-preserving malware detection techniques that can safeguard user privacy while effectively detecting and mitigating malware threats. This includes exploring cryptographic protocols, secure multi-party computation, and privacy-enhancing technologies to enable malware mitigation without compromising the confidentiality of user data.

BIBLIOGRAPHY

[1] A. A. Elkhail and T. Cerny, "On relating code smells to security vulnerabilities," in *2019 IEEE 5th intl conference on big data security on cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE intl conference on intelligent data and security (IDS)*. IEEE, 2019, pp. 7–12.

[2] A. A. Elkhail, J. Svacina, and T. Cerny, "Intelligent token-based code clone detection system for large scale source code," in *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, 2019, pp. 256–260.

[3] A. A. Elkhail, N. Lachtar, D. Ibdah, R. Aslam, H. Khan, A. Bacha, and H. Malik, "Seamlessly safeguarding data against ransomware attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 1–16, 2023.

[4] N. Lachtar, A. A. Elkhail, A. Bacha, and H. Malik, "A cross-stack approach towards defending against cryptojacking," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 126–129, 2020.

[5] ——, "An application agnostic defense against the dark arts of cryptojacking," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 314–325.

[6] D. Ibdah, N. Lachtar, A. A. Elkhail, A. Bacha, and H. Malik, "Dark firmware: A systematic approach to exploring application security risks in the presence of untrusted firmware." in *RAID*, 2020, pp. 413–426.

[7] A. A. Elkhail, R. U. D. Refat, R. Habre, A. Hafeez, A. Bacha, and H. Malik, "Vehicle security: A survey of security issues and vulnerabilities, malware attacks and defenses," *IEEE Access*, vol. 9, pp. 162 401–162 437, 2021.

[8] R. U. D. Refat, A. A. Elkhail, and H. Malik, "Machine learning for automotive cybersecurity: Challenges, opportunities and future directions," *AI-enabled Technologies for Autonomous and Connected Vehicles*, pp. 547–567, 2022.

[9] ——, "A lightweight intrusion detection system for can protocol using neighborhood similarity," in *2022 7th International Conference on Data Science and Machine Learning Applications (CDMA)*. IEEE, 2022, pp. 121–126.

[10] R. U. D. Refat, A. A. Elkhail, A. Hafeez, and H. Malik, "Detecting can bus intrusion by applying machine learning method to graph based features," in *Intelligent Systems and Applications: Proceedings of the 2021 Intelligent Systems Conference (IntelliSys) Volume 3*. Springer, 2022, pp. 730–748.

[11] E. Shakshuki, A. A. Elkhail, I. Nemer, M. Adam, and T. Sheltami, "Comparative study on range free localization algorithms," *Procedia Computer Science*, vol. 151, pp. 501–510, 2019.

[12] I. Nemer, T. Sheltami, E. Shakshuki, A. A. Elkhail, and M. Adam, "Performance evaluation of range-free localization algorithms for wireless sensor networks," *Personal and Ubiquitous Computing*, vol. 25, pp. 177–203, 2021.

[13] A. A. Elkhail and U. Baroudi, "Real-time healthcare monitoring system using smartphones," in *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2018, pp. 1–6.

[14] U. Baroudi, A. Abu Elkhail, and H. Alfares, "Optimum bi-level hierarchical clustering for wireless mobile tracking systems," *Wireless Networks*, vol. 26, pp. 1227–1242, 2020.

[15] A. A. Elkhail, U. Baroudi, and M. Younis, "Wsn routing protocols: Anonymity prospective analysis," in *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE, 2022, pp. 819–823.

[16] H. K. Alfares, A. Abu Elkhail, and U. Baroudi, "Iterative clustering for energy-efficient large-scale tracking systems," *Wireless Personal Communications*, vol. 110, pp. 713–733, 2020.

[17] A. Abuelkhail, U. Baroudi, M. Raad, and T. Sheltami, "Internet of things for healthcare monitoring applications based on rfid clustering scheme," *Wireless Networks*, vol. 27, pp. 747–763, 2021.

[18] A. A. Elkhail, U. Baroudi, and H. Alfares, "System and method for hierarchical clustering of wireless mobile network," Aug. 9 2022, uS Patent 11,412,039.

[19] A. A. Elkhail and U. Baroudi, "Internet of things for healthcare monitoring applications based on rfid clustering scheme," Nov. 10 2020, uS Patent 10,834,549.

[20] U.S. Goverment Interagency, "How to Protect Your Networks from Ransomware," https://www.us-cert.gov/sites/default/files/publications/Ransomware-Executive-One-Pager-and-Technical-Document-FINAL.pdf.

[21] M. Farber, "Global Ransomware Damage Costs To Exceed $265 Billion By 2031," 2017, https://www.einnews.com/pr_news/542950077/global-ransomware-damage-costs-to-exceed-265-billion-by-2031.

[22] A. Greenberg, "Ransomware Attacks Hit Manufacturing - Are You Vulnerable?" 2018, https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world.

[23] Y. Shohet, "Ransomware Attacks Hit Manufacturing - Are You Vulnerable?" 2019, https://www.industryweek.com/technology-and-iiot/ransomware-attacks-hit-manufacturing-are-you-vulnerable.

[24] T. Spring, "Colonial Pipeline Shells Out $5M in Extortion Payout, Report," 2021, https://threatpost.com/colonial-pays-5m/166147/.

[25] G. Torbet, "Baltimore ransomware attack will cost the city over $18 million," 2019, https://www.engadget.com/2019/06/06/baltimore-ransomware-18-million-damages.

[26] S. Morgan, "Global Ransomware Damage Costs Predicted To Reach $20 Billion (USD) By 2021," 2019, https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-20-billion-usd-by-2021.

[27] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "{UNVEIL}: A large-scale, automated approach to detecting ransomware," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 757–772.

[28] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "Shieldfs: a self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 336–347.

[29] J. Chen, C. Wang, Z. Zhao, K. Chen, R. Du, and G.-J. Ahn, "Uncovering the face of android ransomware: Characterization and real-time detection," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1286–1300, 2017.

[30] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, "Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2231–2244.

[31] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "Paybreak: Defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 599–611.

[32] N. Scaife, H. Carter, P. Traynor, and K. R. Butler, "Cryptolock (and drop it): stopping ransomware attacks on user data," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2016, pp. 303–312.

[33] B. Jethva, I. Traoré, A. Ghaleb, K. Ganame, and S. Ahmed, "Multilayer ransomware detection using grouped registry key operations, file entropy and file signature monitoring," *Journal of Computer Security*, vol. 28, no. 3, pp. 337–373, 2020.

[34] S. Jung and Y. Won, "Ransomware detection method based on context-aware entropy analysis," *Soft Computing*, vol. 22, no. 20, pp. 6731–6740, 2018.

[35] L. Constantin, "Ransomware operators might be dropping file encryption in favor of corrupting files. 2022," https://www.csoonline.com/article/573747, 2022.

[36] C. Neagu, "BlackCat's Ransomware Tool Gets an Upgrade. 2022," https://heimdalsecurity.com/blog/blackcat-ransomware-upgrade/, 2022.

[37] S. Kaba, "Exmatter Points to Potential Future of Data Extortion. 2022," https://www.cyderes.com/blog/threat-advisory-exmatter-data-extortion/, 2022.

[38] D. Mayer, "Exmatter: Clues to the future of data extortion. 2022," https://stairwell.com/news/threat-research-report-exmatter-future-of-data-extortion/, 2022.

[39] L. Tung, "MongoDB ransacked: Now 27,000 databases hit in mass ransom attacks. 2017," https://www.zdnet.com/article/mongodb-ransacked-now-27000-databases-hit-in-mass-ransom-attacks/, 2017.

[40] B. Durbin, "BlackMatter ransomware gang says it's shutting down over law enforcement pressure. 2021," https://techcrunch.com/2021/11/03/blackmatter-ransomware-shut-down/, 2021.

[41] D. Milmo, "Ransomware payments nearly double in one year. 2023," https://amp.theguardian.com/technology/2023/may/10/ransomware-payments-nearly-double-in-one-year/, 2023.

[42] K. P. Subedi, D. R. Budhathoki, B. Chen, and D. Dasgupta, "Rds3: Ransomware defense strategy by using stealthily spare space," in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2017, pp. 1–8.

[43] M. Baykara and B. Sekin, "A novel approach to ransomware: Designing a safe zone system," in *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*. IEEE, 2018, pp. 1–5.

[44] L. Ceci, "Number of mobile app downloads worldwide from 2016 to 2022," https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads, 2023.

[45] "Mobile Operating System Market Share Worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide, 2022.

[46] A. Friedman, "Google kept over a million bad apps out of the Play Store last year," https://www.phonearena.com/news/google-kept-more-than-million-bad-apps-out-of-playstore, 2022.

[47] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[48] I. Vijonivic, "Ransomware statistics in 2023: From random barrages to targeted hits," https://dataprot.net/statistics/ransomware-statistics/, 2023.

[49] C. Anderson, "CovidLock Update: Deeper Analysis of Coronavirus Android Ransomware," https://www.domaintools.com/resources/blog/covidlock-update-coronavirus-ransomware/, 2020.

[50] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 651–661.

[51] D. Sgandurra, L. Muñoz-González, R. Mohsen, and E. C. Lupu, "Automated dynamic analysis of ransomware: Benefits, limitations and use for detection," *arXiv preprint arXiv:1609.03020*, 2016.

[52] H. Zhang, X. Xiao, F. Mercaldo, S. Ni, F. Martinelli, and A. K. Sangaiah, "Classification of ransomware families with machine learning based onn-gram of opcodes," *Future Generation Computer Systems*, vol. 90, pp. 211–221, 2019.

[53] S. Song, B. Kim, and S. Lee, "The effective ransomware prevention technique using process monitoring on android platform," *Mobile Information Systems*, vol. 2016, 2016.

[54] A. Alzahrani, A. Alshehri, H. Alshahrani, R. Alharthi, H. Fu, A. Liu, and Y. Zhu, "Randroid: Structural similarity approach for detecting ransomware applications in android platform," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0892–0897.

[55] N. Andronio, S. Zanero, and F. Maggi, "Heldroid: Dissecting and detecting mobile ransomware," in *international symposium on recent advances in intrusion detection*. Springer, 2015, pp. 382–404.

[56] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-packdroid: Api package-based characterization and detection of mobile ransomware," in *Proceedings of the symposium on applied computing*, 2017, pp. 1718–1723.

[57] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[58] S. Alsoghyer and I. Almomani, "Ransomware detection system for android applications," *Electronics*, vol. 8, no. 8, p. 868, 2019.

[59] H. Faris, M. Habib, I. Almomani, M. Eshtay, and I. Aljarah, "Optimizing extreme learning machines using chains of salps for efficient android ransomware detection," *Applied Sciences*, vol. 10, no. 11, p. 3706, 2020.

[60] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2017, pp. 252–276.

[61] B. I. S. Office, "Frequently Asked Questions - Ransomware," https://security.berkeley.edu/faq/ransomware.

[62] R. Love, *Linux kernel development*. Pearson Education, 2010.

[63] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[64] L. E. Bassham III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks *et al.*, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.

[65] Kaspersky, "What is WannaCry ransomware?" https://usa.kaspersky.com/resource-center/threats/ransomware-wannacry.

[66] L. Abrams, "Lorenz ransomware decryptor recovers victims' files for free. 2021," https://www.bleepingcomputer.com/news/security/lorenz-ransomware-decryptor-recovers-victims-files-for-free/, 2021.

[67] B. Toulas, "Avast releases free decryptor for mafiaware666 ransomware variants. 2022."

[68] A. Chebac, "Data Corruption, A Potential New Trend in Ransomware Attacks. 2022," https://heimdalsecurity.com/blog/data-corruption-a-potential-new-trend-in-ransomware-attacks/, 2022.

[69] C. G. S. . C. S. R. Teams, "Royal Rumble: Analysis of Royal Ransomware. 2022," https://www.cybereason.com/blog/royal-ransomware-analysis/, 2022.

[70] A. Drapkin, "This Ransomware Is Actually Malware That Will Delete Your Files. 2022," https://tech.co/news/ransomware-actually-malware-will-delete-your-files/, 2022.

[71] M. Loman, "How the most damaging ransomware evades IT security," https://news.sophos.com/en-us/2019/11/14/how-the-most-damaging-ransomware-evades-it-security, 2019.

[72] M. Nadeau, "11 ransomware trends for 2018. 2018," https://www.csoonline.com/article/3267544/11-ways-ransomware-is-evolving.html/, 2018.

[73] R. Lipovskỳ, L. Štefanko, and G. Braniša, "The rise of android ransomware," 2016.

[74] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[75] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[76] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.

[77] K. Lee, S.-Y. Lee, and K. Yim, "Machine learning based file entropy analysis for ransomware detection in backup systems," *IEEE Access*, vol. 7, pp. 110 205–110 215, 2019.

[78] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 921–937.

[79] Y. Takeuchi, K. Sakai, and S. Fukumoto, "Detecting ransomware using support vector machines," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, 2018, pp. 1–6.

[80] R. Vinayakumar, K. Soman, K. S. Velan, and S. Ganorkar, "Evaluating shallow and deep networks for ransomware detection and classification," in *2017 international conference on advances in computing, communications and informatics (ICACCI)*. IEEE, 2017, pp. 259–265.

[81] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the gordian knot: A look under the hood of ransomware attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 3–24.

[82] C. Moore, "Detecting ransomware with honeypot techniques," in *2016 Cybersecurity and Cyberforensics Conference (CCC)*. IEEE, 2016, pp. 77–81.

[83] A. Kharraz and E. Kirda, "Redemption: Real-time protection against ransomware at end-hosts," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 98–119.

[84] S. Mehnaz, A. Mudgerikar, and E. Bertino, "Rwguard: A real-time detection system against cryptographic ransomware," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 114–136.

[85] J. Baldwin and A. Dehghantanha, "Leveraging support vector machine for opcode density based detection of crypto-ransomware," in *Cyber threat intelligence*. Springer, 2018, pp. 107–136.

[86] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.

[87] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 457–468.

[88] Z.-G. Chen, H.-S. Kang, S.-N. Yin, and S.-R. Kim, "Automatic ransomware detection and analysis based on dynamic api calls flow graph," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, 2017, pp. 196–201.

[89] S. Homayoun, A. Dehghantanha, M. Ahmadzadeh, S. Hashemi, and R. Khayami, "Know abnormal, find evil: frequent pattern mining for ransomware threat hunting and intelligence," *IEEE transactions on emerging topics in computing*, vol. 8, no. 2, pp. 341–351, 2017.

[90] S. K. Shaukat and V. J. Ribeiro, "Ransomwall: A layered defense system against cryptographic ransomware attacks using machine learning," in *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*. IEEE, 2018, pp. 356–363.

[91] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *computers & security*, vol. 77, pp. 578–594, 2018.

[92] N. Lachtar, D. Ibdah, and A. Bacha, "Toward mobile malware detection through convolutional neural networks," *IEEE Embedded Systems Letters*, vol. 13, no. 3, pp. 134–137, 2020.

[93] ——, "The case for native instructions in the detection of mobile ransomware," *IEEE Letters of the Computer Society*, vol. 2, no. 2, pp. 16–19, 2019.

[94] G. Sharma, A. Johri, A. Goel, A. Gupta *et al.*, "Enhancing ransomwareelite app for detection of ransomware in android applications," in *2018 Eleventh International Conference on Contemporary Computing (IC3)*. IEEE, 2018, pp. 1–4.

[95] C. Zheng, N. Dellarocca, N. Andronio, S. Zanero, and F. Maggi, "Greateatlon: Fast, static detection of mobile ransomware," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2016, pp. 617–636.

[96] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *International symposium on research in attacks, intrusions, and defenses*. Springer, 2017, pp. 192–214.

[97] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2016.

[98] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for android ransomware," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 2015, pp. 1338–1343.

[99] A. Gharib and A. Ghorbani, "Dna-droid: A real-time android ransomware detection framework," in *International Conference on Network and System Security*. Springer, 2017, pp. 184–198.

[100] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[101] N. Lachtar, D. Ibdha, H. Khan, and A. Bacha, "Ransomshield: A visualization approach to defending mobile systems against ransomware," vol. 26, no. 3. New York, NY, USA: Association for Computing Machinery, 2023, p. 30. [Online]. Available: https://doi.org/10.1145/3579822

[102] H. Kim, D. Yoo, J.-S. Kang, and Y. Yeom, "Dynamic ransomware protection using deterministic random bit generator," in *2017 IEEE Conference on Application, Information and Network Security (AINS)*. IEEE, 2017, pp. 64–68.

[103] J.-Y. Paik, K. Shin, and E.-S. Cho, "Poster: Self-defensible storage devices based on flash memory against ransomware," in *Proceedings of IEEE Symposium on Security and Privacy*, 2016.

[104] S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang, "Ssd-insider: internal defense of solid-state drive against ransomware with perfect data recovery," in

2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2018, pp. 875–884.

[105] D. Min, D. Park, J. Ahn, R. Walker, J. Lee, S. Park, and Y. Kim, "Amoeba: an autonomous backup and recovery ssd for ransomware attack defense," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 245–248, 2018.

[106] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped ssds with flashmap," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015, pp. 580–591.

[107] G. Hung and M. Joven, "Petya's Master Boot Record Infection," 2017, https://www.fortinet.com/blog/threat-research/petya-s-master-boot-record-infection.

[108] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[109] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 101–111.

[110] J. Axboe *et al.*, "Flexible i/o tester," *Online] https://github. com/axboe/fio*, 2016.

[111] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *USENIX; login*, vol. 41, no. 1, pp. 6–12, 2016.

[112] M. Labs, "Petya – Taking Ransomware To The Low Level," https://blog.malwarebytes.com/threat-analysis/2016/04/petya-ransomware.

[113] D. Olenick, "How Conti Ransomware Works," https://www.bankinfosecurity.com/how-conti-ransomware-works-a-15763, 2021.

[114] K. Cabaj, M. Gregorczyk, and W. Mazurczyk, "Software-defined networking-based crypto ransomware detection using http traffic characteristics," *Computers & Electrical Engineering*, vol. 66, pp. 353–368, 2018.

[115] C. Cimpanu, " Conti Ransomware Attack. 2020 ," https://www.zdnet.com/article/, 2021.

[116] C. Sandbox, "Cuckoo sandbox: automated malware analysis," 2015.

[117] "VirusTotal," https://www.virustotal.com/gui/home/upload.

[118] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[119] "Netperf." https://hewlettpackard.github.io/netperf/.

[120] "PassMark PerformanceTest Tool," https://play.google.com/store/apps/details?id=com.passmark.pt_mobile&hl=en_US&gl=US&pli=1.

[121] E. Vanderburg, "The evolution of a cybercrime: A timeline of ransomware advances," https://www.carbonite.com/blog/article/2017/08/the-evolution-of-a-cybercrime-a-timeline-of-ransomware-advances, 2017.

[122] S. Khandelwal, "LockerPin Ransomware Resets PIN and Permanently Locks Your SmartPhones," https://thehackernews.com/2015/09/android-lock-ransomware.html, 2015.

[123] N. Lachtar, D. Ibdah, H. Khan, and A. Bacha, "Ransomshield: A visualization approach to defending mobile systems against ransomware," *ACM Transactions on Privacy and Security*, vol. 26, no. 3, pp. 1–30, 2023.

[124] J. Morris, E. Lifland, J. Y. Yoo, J. Grigsby, D. Jin, and Y. Qi, "Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 119–126.

[125] J. J. Webster and C. Kit, "Tokenization as the initial phase in nlp," in *COLING 1992 volume 4: The 14th international conference on computational linguistics*, 1992.

[126] H. P. Luhn, "A statistical approach to mechanized encoding and searching of literary information," *IBM Journal of research and development*, vol. 1, no. 4, pp. 309–317, 1957.

[127] C. Silva and B. Ribeiro, "The importance of stop word removal on recall values in text categorization," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, vol. 3. IEEE, 2003, pp. 1661–1666.

[128] P. Halácsy and V. Trón, "Benefits of deep nlp-based lemmatization for information retrieval." in *CLEF (Working Notes)*, 2006.

[129] Y. Kim, "Convolutional neural networks for sentence classification," 2014.

[130] F. Pineda, "Generalization of back propagation to recurrent and higher order neural networks," in *Neural information processing systems*, 1987.

[131] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[132] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[133] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[134] T. TensorFlow, "An open source machine learning framework for everyone," 2019.

[135] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[136] "Tesseract-OCR," https://tesseract-ocr.github.io/.

[137] "GoogleTrans," https://pypi.org/project/googletrans/.

[138] J. Gao, J. Lanchantin, M. L. Soffa, and Y. Qi, "Black-box generation of adversarial text sequences to evade deep learning classifiers," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 50–56.

[139] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating natural language adversarial examples," *arXiv preprint arXiv:1804.07998*, 2018.

[140] S. Ren, Y. Deng, K. He, and W. Che, "Generating natural language adversarial examples through probability weighted word saliency," in *Proceedings of the 57th annual meeting of the association for computational linguistics*, 2019, pp. 1085–1097.

[141] M. H. Samavatian, A. Bacha, L. Zhou, and R. Teodorescu, "Rnnfast: An accelerator for recurrent neural networks using domain-wall memory," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 4, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3399670