Seamlessly Safeguarding Data Against Ransomware Attacks

Abdulrahman Abu Elkhail, *Member, IEEE*, Nada Lachtar, *Member, IEEE*, Duha Ibdah, *Member, IEEE*, Rustam Aslam, Hamza Khan, Anys Bacha *Member, IEEE*, and Hafiz Malik, *Member, IEEE*

Abstract—Encryption has become an indispensable technology for preserving confidentiality. Unfortunately, cybercriminals have re-purposed this technology to deny users access to their data. This trend has sparked an onslaught of ransomware attacks, that resulted in several victims being extorted to pay ransoms in return for restoring their maliciously encrypted data. In response to these challenges, we propose a novel runtime solution that seamlessly defends against cryptographic ransomware. A key observation made by this work is that maliciously encrypted data is initially buffered in the OS'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 ransomware samples and show that our design reliably restores all encrypted files. Furthermore, our solution is resilient to ransomware that employ techniques including master boot record infection and multi-threaded attacks. Finally, an evaluation of our proof-of-concept implementation shows minimal performance impact while running a mix of compute and I/O bound applications.

Index Terms—Ransomware, malware, data recovery, encryption, system security, intrusion detection system.

1 Introduction

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 [53]. 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 [11]. 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 [12], [47]. 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 [48]. 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 [52]. 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 [36]. 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 [8], [10], [16], [18], [19], [21], [26], [44]. A large portion of this research focused on the detection of such malware. For example, Kharraz et al. [21] employed a temporary environment designed to screen user programs. Other solutions [18], [19], [44] 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 [10], [16], [26]. Huang et al. [16] 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 [26] 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 intro-

The authors are with the University of Michigan, Dearborn, MI, 48128.

This work was supported in part by the National Science Foundation under grant CNS-1947580.

duces, ransomware can bypass this defense by using their own crypto libraries over OS hosted services. In general, recovery solutions such as [16], [26] require manual user intervention where victims can experience long delays when restoring compromised data. Similarly, recovery solutions that employ backups [10] 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.

This paper proposes 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 [17] and multi-threaded attacks [31]. 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 [6], SPLASH-3 [42], Flexible I/O [2], and Filebench [51], show an average performance degradation that is less than 2% across both compute and I/O bound workloads.

Overall, this paper 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 paper is organized as follows: Section 2 presents background information. Section 3 discusses the threat model. Section 4 describes the design of the proposed system. Section 5 presents the methodology and experimental framework used in this work. Section 6 discusses the results of our evaluation. Section 8 details related work; and Section 9 concludes.

2 Background and Motivation

2.1 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) [32].

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.

2.2 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 [39]. 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 [5]. 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 (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.

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

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 [20]. We executed a sample of the aforementioned ransomware and monitored its disk 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 1.

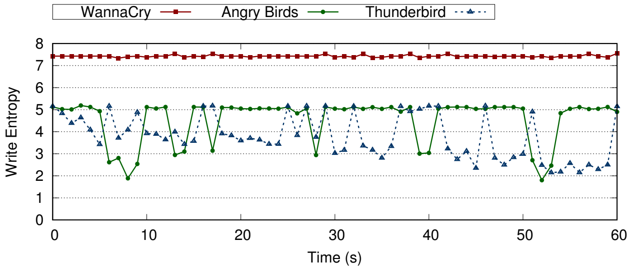


Fig. 1: Entropy of data written to storage as a function of time for WannaCry ransomware, Angry Birds, and Thunderbird.

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. We discuss more extensive results of this metric in section 6.

3 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 [27].

- **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 [31], [38].

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 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.

4.1 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 [7]. 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 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.

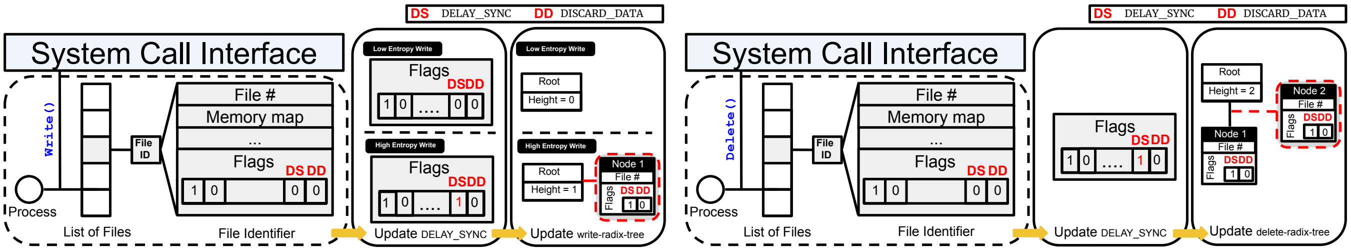
Write Requests. Whenever a process issues a write request to the kernel via the `write()` system call, our solution computes the entropy of the data that is to be written as a first step. Further actions are then taken based on the outcome of this computation. In the event that the entropy of the computed data exceeds a programmable threshold, the system proceeds to updating the `tgid_ransom_t` data structure (Listing 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 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 synchroniz-



(a) Write

(b) Delete

Fig. 2: Overview of the system call interface and how different system calls update the tags of their radix trees.

ing the marked file to the disk until further assessment is made about its corresponding 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 2(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 2(b).

4.2 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 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 determine 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 3.

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

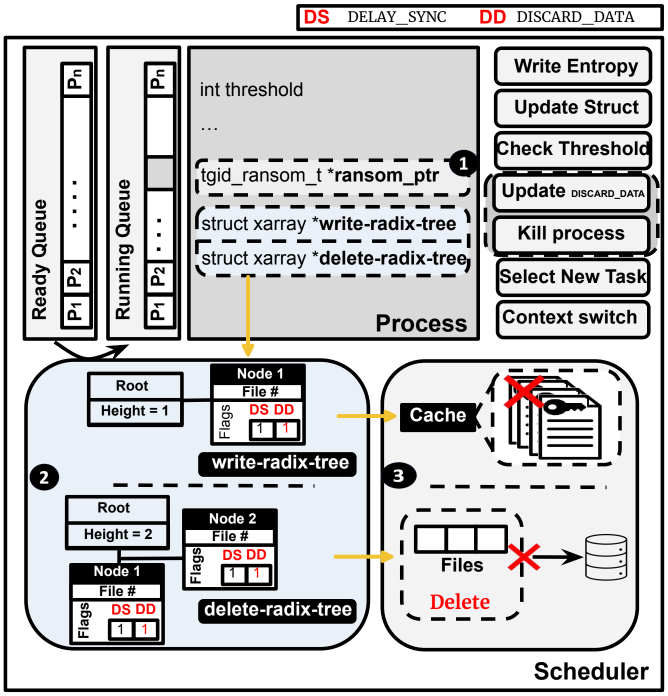


Fig. 3: Overview of the OS scheduler design.

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 [9], [31], [38] 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 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 structures (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 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 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 [27] 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.

5 Methodology

We conducted ransomware experiments using the Cuckoo sandbox [43]. 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 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 [1] repository. Our dataset included both MBR and multi-threaded ransomware. The samples we used, their families, and respective capabilities are listed in Table 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 7 which can be found in the Appendix.

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 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 [6] and SPLASH-3 [42] 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 [2] and Filebench [51] 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 2. The configuration of the respective I/O benchmarks is outlined in Tables 3 and 4.

6 Evaluation

6.1 Ransomware Analysis

6.1.1 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 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.

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 [13] 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.

6.1.2 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 1.

Overall, our results show that our solution is effective in dynamically distinguishing 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 5.

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

Family	Samples	Algorithm	Backup Spoliation	MBR	Multi-threaded	Defeated	Discovered Month
7ev3n	9	RSA+AES-256	✓	✗	✗	✓	2016-4
Cerber	46	RSA+RC4-256	✓	✗	✓	✓	2016-3
Conti	57	RSA+AES-256	✓	✗	✓	✓	2019-12
Crypmod	132	RSA+AES-256	✓	✗	✗	✓	2018-2
Crypren	5	RSA+AES-256	✓	✗	✗	✓	2016-1
CryptoWall	158	RSA+AES-256	✓	✗	✗	✓	2013-12
DeriaLock	2	RSA+AES-256	✓	✗	✗	✓	2017-12
Dharma	3	RSA+AES-256	✓	✗	✗	✓	2016-1
InfinityCrypt	2	RSA+AES-256	✗	✗	✗	✓	2017-9
Locky	351	RSA+AES-128	✓	✗	✗	✓	2016-1
Maktub	3	RSA+AES-256	✓	✗	✗	✓	2016-3
Rapid	4	RSA+AES-256	✗	✗	✗	✓	2017-12
Petya	4	RSA+AES-256	✓	✓	✗	✓	2016-3
RedEye	2	RSA+AES-256	✓	✓	✗	✓	2018-6
Shade	13	RSA+AES-256	✓	✗	✗	✓	2014-12
SporaRansomware	6	RSA+AES-256	✓	✗	✗	✓	2017-1
TeslaCrypt	501	ECC+AES-256	✓	✗	✗	✓	2016-1
WannaCry	26	RSA+AES-256	✓	✗	✗	✓	2017-5
Total Samples	**1324**						

TABLE 1: Summary of ransomware families and their capabilities.

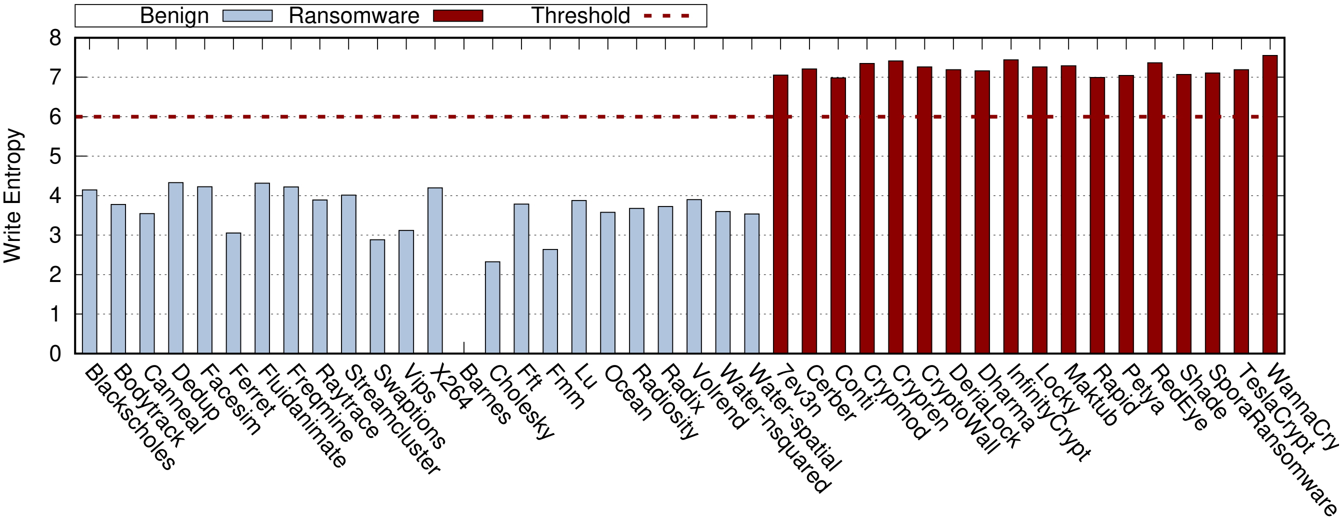


Fig. 4: A comparison of write entropy measurements across PARSEC, SPLASH-3, and ransomware families over a one second execution period.

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 2: Summary of performance benchmarks.

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`.

6.2 Performance Overhead

6.2.1 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

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 3: 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: Configuration of the Filebench workloads.

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

TABLE 5: Summary of false positive and negative results.

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 5 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 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.

6.2.2 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 6. We observed that most of the applications from the Filebench suite experienced a slight decrease relative to the original design after being configured with the parameters shown in Table 4. 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 6(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 6(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 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 7. 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 8. 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

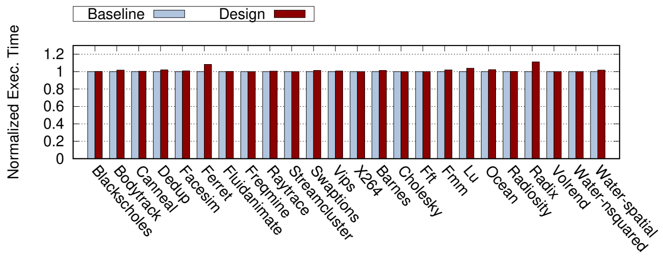


Fig. 5: Performance impact of our design on compute-bound workloads relative to an unsecured baseline.

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.

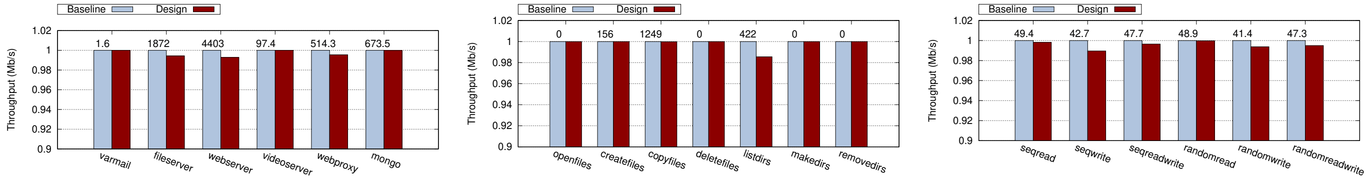
7 Discussion and Limitations

A primary detection feature that our proof of concept involves relates to measuring the amount of entropy inherent in write transactions. Although our evaluation shows that the aforementioned metric is effective in distinguishing between ransomware and most benign applications, it still suffers from misclassifying a small set of workloads. For instance, compressed data tends to exhibit entropy levels that are comparable to information that undergoes encryption. A user may also choose to encrypt a set of personal files on the system which would naturally trigger a false positive. As such, the entropy metric alone is not sufficient for differentiating between ransomware and compression utilities. Consequently, our design harnesses other features that include the monitoring of network activity and file deletion requests as a means of reducing the number of false positives. Our evaluation shows that while considering the aforementioned supplemental features, our design did not classify any of the 32 cryptographic and compression applications listed in Table 7 as malicious. However, as a conservative measure, we still inform the user when such utilities are running and present it with the option to approve or deny their execution. The user can also white-list such applications, so future runs do not prompt the user for action.

Multiple studies explored different features that could also be used for fingerprinting ransomware activity [10], [21], [22], [35], [44]. Such features include the use of decoy files [10], [35], the monitoring of directory traversal patterns [22], file type conversions [44], in addition to entropy levels of written data [21]. However, unlike prior detection mechanisms that leave filesystems partially encrypted or require delays for recovering data, our solution dynamically preserves user data without additional backups or manual intervention. As such, our work complements these detection-based studies by focusing on the restoration aspect of data already impacted by ransomware. Although our solution accommodates the integration of the aforementioned features, we leverage entropy, network requests, and delete operations as features for detecting maliciously encrypted data as part of our proof of concept design. We believe our end-to-end solution significantly raises the bar for attackers. For instance, our evaluation demonstrates that harnessing entropy, network requests, and delete operations as features resulted in our solution detecting all ransomware programs within our dataset, which included over one thousand samples. Most importantly, our design was able to reliably restore all impacted files while tolerating malicious techniques, such as master boot record infection and multi-threaded attacks. Furthermore, our proof-of-concept implementation shows that our solution introduces minimal performance impact while running a mix of compute and I/O bound applications.

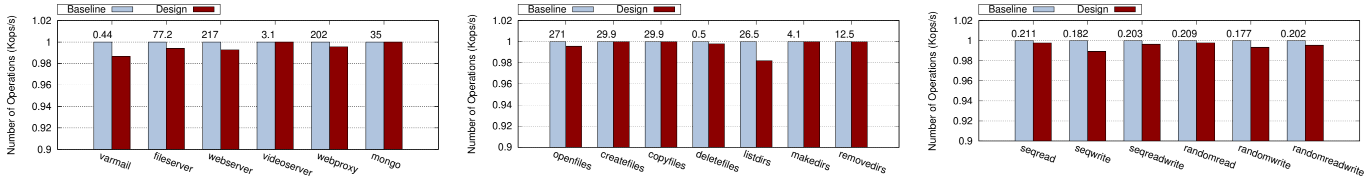
Despite the robustness of our solution against more than one thousand ransomware samples, its ability to reliably restore impacted files, as well as tolerate malicious techniques, such as master boot record infection attacks, attackers may attempt to evade our defense through smart ransomware. Therefore, to understand the limitations of our solution in this context, we developed a synthetic workload that encrypted files in a throttled fashion. We found that our solution was able to detect throttled write operations that employed data rates as low as 512 B/s. Although smart ransomware may attempt to evade our defense by writing data using rates that are well below 512 B/s, we argue that ransomware often aims for speed when encrypting files in order to outpace any human response. For instance, our experiments show that write transactions issued by ransomware sustained data rates that ranged between 41 MB/s – 147 MB/s. This is $8.4 \cdot 10^4$x – $3 \cdot 10^5$x higher than the rate our solution was able to detect. To address such throttling attacks, our defense buffers written data until a minimum of 512 bytes have been issued, after which the entropy is computed. Meanwhile, the process is periodically evaluated by the scheduler. Once a minimum of 512 bytes have been written by a given process, the scheduler examines the process's `socket_created` and `delete_requested` fields in conjunction with the computed entropy to classify the running process as either benign, malicious, or suspicious. Additionally, ransomware may choose to overwrite the files with null bytes or shuffle their content and make them unreadable [37]. Although we haven't encountered any samples that performed the aforementioned actions, designing ransomware with such characteristics is still possible and would evade our system.

In addition to throttling write transactions, ransomware may evade detection by distributing its execution across multiple cores using different threads [31], [38]. Our design can detect such attacks since it uses an application granular approach. For instance, all of the entropy, network, and deletion information is aggregated into a common data structure (`tgid_ransom_t`) irrespective of how many threads the ransomware uses. Furthermore, ransomware may employ delayed attacks through the insertion of stalling code [25]. Our design can defend against such attacks since every



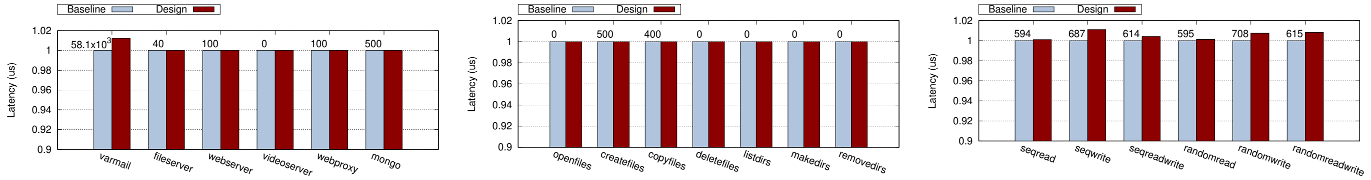
(a) Filebench (b) File operations (c) FIO

Fig. 6: Summary of throughput experiments as a function of macro and micro benchmarks from Filebench and FIO.



(a) Filebench (b) File operations (c) FIO

Fig. 7: Summary of IOPS experiments as a function of macro and micro benchmarks from Filebench and FIO.



(a) Filebench (b) File operations (c) FIO

Fig. 8: Summary of latency experiments as a function of macro and micro benchmarks from Filebench and FIO.

running process is periodically evaluated for maliciousness through the OS scheduler. If a process is initially classified as benign during a given evaluation cycle, the scheduler will continue to periodically monitor the process and flag it in the cycle that correlates to the malware disclosing its malicious activity. Finally, sophisticated forms of ransomware may exploit system level vulnerabilities and carry out a privilege escalation attack prior to encrypting the victim's data. Although our design can still detect malicious processes that run as root, once a process gains root access, it could disable our defense through a newly installed kernel and in turn compromise the victim's data. However, we argue that any in-host defense would be defeated under such assumptions. Furthermore, most ransomware nowadays run in user mode since this is sufficient to encrypt the victim's data [31].

8 Related Work

Ransomware Detection. A large body of research [8], [19], [21], [30], [44], [55] explored various detection techniques that are aimed at defending against cryptographic ransomware. For example, Kharraz et al. [21] 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 [10], [23], [35]. More recent work [22], [33] expanded on the aforementioned approach by monitoring a range of content and behavior-based features for fingerprinting ransomware activity. Other work by Scaife et al. [44] 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 [18], [19] 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 [35] 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. [10] 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 [4], [14], [28], [29], [41], [45], [46], [50], [54], [56]. For example, EldeRan [45], a solution proposed by Sgandurra et al., leveraged machine learning to infer ransomware activity based on early actions taken during the installation process. On the other hand, work by Takeuchi et al. [50] and Vinayakumar et al. [54] proposed a detection scheme that used API calls as features. More recent work [4], [56], on the other hand, considered the use of opcode sequences, instead, as features

TABLE 6: Comparison of our solution with existing ransomware defense systems.

Solution	Dataset Size (Crypto Families)	FP	FN	Filesystem Activity	Network Activity	Process Activity	MBR Detection Capability	File Recovery	Real-time Recovery	I/O Performance Overhead
UNVEIL [21]	319 (8)	0.0%	0.0%	✓	✗	✓	✗	✗	✗	-
Redemption [22]	677 (29)	0.8%	0.0%	✓	✗	✓	✗	✗	✗	5.6%
CryptoDrop [44]	492 (15)	3%	0.0%	✓	✗	✓	✗	✗	✗	-
EldeRan [45]	582 (11)	1.6%	3.7%	✗	✗	✓	✗	✗	✗	-
RWGuard [33]	261 (14)	0.1%	0.0%	✓	✗	✓	✗	✓(partial)	✗	1.9%
ShieldFS [10]	305 (11)	0.0%	2%	✗	✗	✓	✗	✓	✗	30%−380%
PayBreak [26]	107 (20)	-	-	✗	✗	✗	✗	✓(partial)	✗	150%
FlashGuard [16]	1,477 (13)	-	-	✗	✗	✗	✗	✓	✗	< 6%
Our Work	**1,324 (18)**	**0.0%**	**0.0%**	✓	✓	✓	✓	✓	✓	< **1%**

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 encrypted filesystems. 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.

Data Recovery. To address the shortcomings of prior work, various researchers proposed different data recovery techniques as a solution against encryption ransomware [3], [15], [16], [24], [26], [34], [40], [49]. For instance, Kolodenker et al. [26] 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. [49] 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 [3], [15], [16], [34], [40], focused on harnessing the intrinsic properties of flash drives. For example, work by Huang et al. [16] 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 data. Similarly, recovery solutions that employ backups [10] 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 data 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.

Comparison to Existing Defenses. We compared our solution 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 6. In general, our evaluation consisted of executing a larger number of cryptographic ransomware samples relative to other work [10], [21], [22], [26], [33], [44], [45]. With the exception of [16], our dataset size for cryptographic ransomware is 2× – 12× larger than what was considered in the other studies. Although FlashGuard [16] used a slightly larger dataset size of 1,477 samples, the study only covered a total of 13 ransomware families. Unlike Flashguard [16], our work considers samples across 18 recently released 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 or negatives throughout the evaluation. The only work that had similar results is UNVEIL [21]. However, unlike UNVEIL [21], our work considers over 4× the amount of cryptographic ransomware samples and covers more than 2× the number of families. The remaining studies [10], [22], [33], [44], [45] 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 activity at the filesystem, process, and network levels. Although other work considered using a combination of filesystem and process activity monitoring for detecting ransomware, none of the studies listed in Table 6 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 6 demonstrated the ability to protect against such malware. In addition to MBR infecting ransomware, we compared our solution to other defenses in terms of file 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 1,324 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 [33] and PayBreak [26] offer file recovery, this recovery is dependant on ransomware using the Microsoft CryptoAPI. As such, the aforementioned solutions cannot recover files that are encrypted by ransomware that employ custom-written cryptographic libraries. Solutions, such as ShieldFS [10] and FlashGuard [16], on the other hand, offer full recovery of impacted files. 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 files 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 6. 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 6. However, we note that multiple studies including [10], [33], [45] employ machine learning algorithms as part of their detection. Such algorithms typically involve a non-trivial amount of computation.

9 Conclusion

This paper presents a novel runtime defense against cryptographic ransomware. 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 ransomware samples that span 18 ransomware families. Furthermore, we demonstrate that our solution is resilient to ransomware that employ techniques including master boot record infection and multi-threaded attacks. Finally, we show that our proof-of-concept implementation incurs negligible overhead while running a diverse set of workloads.

Acknowledgments

The authors would like to thank the anonymous reviewers and the editor for their feedback and comments on this work. This work was funded in part by the National Science Foundation under grant CNS-1947580.

References

[1] VirusTotal. https://www:virustotal:com.

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

[3] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun 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)*, pages 875–884. IEEE, 2018.

[4] James Baldwin and Ali Dehghantanha. Leveraging support vector machine for opcode density based detection of crypto-ransomware. In *Cyber Threat Intelligence*, pages 107–136. Springer, 2018.

[5] Lawrence E Bassham III, Andrew L Rukhin, Juan Soto, James R Nechvatal, Miles E Smid, Elaine B Barker, Stefan D Leigh, Mark Levenson, Mark Vangel, David L Banks, et al. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications, 2010.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] Krzysztof Cabaj, Marcin Gregorczyk, and Wojciech Mazurczyk. Software-defined networking-based crypto ransomware detection using http traffic characteristics. *Computers & Electrical Engineering*, 66:353–368, 2018.

[8] Jing Chen, Chiheng Wang, Ziming Zhao, Kai Chen, Ruiying Du, and Gail-Joon Ahn. Uncovering the face of android ransomware: Characterization and real-time detection. *IEEE Transactions on Information Forensics and Security*, 13(5):1286–1300, 2017.

[9] Catalin Cimpanu. Conti Ransomware Attack. 2020 . https://www.zdnet.com/article/conti-ransomware-uses-32-simultaneous-cpu-threads-for-blazing-fast-encryption, 2021.

[10] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 336–347, 2016.

[11] Malcomb 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.

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

[13] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[14] Sajad Homayoun, Ali Dehghantanha, Marzieh Ahmadzadeh, Sattar Hashemi, and Raouf Khayami. Know abnormal, find evil: frequent pattern mining for ransomware threat hunting and intelligence. *IEEE transactions on emerging topics in computing*, 8(2):341–351, 2017.

[15] Jian Huang, Anirudh Badam, Moinuddin K Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 580–591, 2015.

[16] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin 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*, pages 2231–2244, 2017.

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

[18] Brijesh Jethva, Issa Traoré, Asem Ghaleb, Karim Ganame, and Sherif Ahmed. Multilayer ransomware detection using grouped registry key operations, file entropy and file signature monitoring. *Journal of Computer Security*, 28(3):337–373, 2020.

[19] Sangmoon Jung and Yoojae Won. Ransomware detection method based on context-aware entropy analysis. *Soft Computing*, 22(20):6731–6740, 2018.

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

[21] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. {UNVEIL}: A large-scale, automated approach to detecting ransomware. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 757–772, 2016.

[22] Amin Kharraz and Engin Kirda. Redemption: Real-time protection against ransomware at end-hosts. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 98–119. Springer, 2017.

[23] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin 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*, pages 3–24. Springer, 2015.

[24] Haeun Kim, Dongchang Yoo, Ju-Sung Kang, and Yongjin Yeom. Dynamic ransomware protection using deterministic random bit generator. In *2017 IEEE Conference on Application, Information and Network Security (AINS)*, pages 64–68. IEEE, 2017.

[25] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296, 2011.

[26] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. Paybreak: Defense against cryptographic ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 599–611, 2017.

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

[28] Nada Lachtar, Duha Ibdah, and Anys Bacha. The case for native instructions in the detection of mobile ransomware. *IEEE Letters of the Computer Society*, 2(2):16–19, 2019.

[29] Nada Lachtar, Duha Ibdah, and Anys Bacha. Toward mobile malware detection through convolutional neural networks. *IEEE Embedded Systems Letters*, 13(3):134–137, 2020.

[30] Kyungroul Lee, Sun-Young Lee, and Kangbin Yim. Machine learning based file entropy analysis for ransomware detection in backup systems. *IEEE Access*, 7:110205–110215, 2019.

[31] Mark 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.

[32] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[33] Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. Rwguard: A real-time detection system against cryptographic ransomware. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 114–136. Springer, 2018.

[34] Donghyun Min, Donggyu Park, Jinwoo Ahn, Ryan Walker, Junghee Lee, Sungyong Park, and Youngjae Kim. Amoeba: an autonomous backup and recovery ssd for ransomware attack defense. *IEEE Computer Architecture Letters*, 17(2):245–248, 2018.

[35] Chris Moore. Detecting ransomware with honeypot techniques. In *2016 Cybersecurity and Cyberforensics Conference (CCC)*, pages 77–81. IEEE, 2016.

[36] Steve 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.

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

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

[39] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[40] Joon-Young Paik, Keuntae Shin, and Eun-Sun Cho. Poster: Self-defensible storage devices based on flash memory against ransomware. In *Proceedings of IEEE Symposium on Security and Privacy*, 2016.

[41] Matilda Rhode, Pete Burnap, and Kevin Jones. Early-stage malware prediction using recurrent neural networks. *computers & security*, 77:578–594, 2018.

[42] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

[43] Cuckoo Sandbox. Cuckoo sandbox: automated malware analysis, 2015.

[44] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. Cryptolock (and drop it): stopping ransomware attacks on user data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 303–312. IEEE, 2016.

[45] Daniele Sgandurra, Luis Muñoz-González, Rabih Mohsen, and Emil C Lupu. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. *arXiv preprint arXiv:1609.03020*, 2016.

[46] Saiyed Kashif Shaukat and Vinay J Ribeiro. Ransomwall: A layered defense system against cryptographic ransomware attacks using machine learning. In *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*, pages 356–363. IEEE, 2018.

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

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

[49] Kul Prasad Subedi, Daya Ram Budhathoki, Bo Chen, and Dipankar Dasgupta. Rds3: Ransomware defense strategy by using stealthily spare space. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2017.

[50] Yuki Takeuchi, Kazuya Sakai, and Satoshi Fukumoto. Detecting ransomware using support vector machines. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, pages 1–6, 2018.

[51] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

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

[53] 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.

[54] R Vinayakumar, KP Soman, KK Senthil Velan, and Shaunak Ganorkar. Evaluating shallow and deep networks for ransomware detection and classification. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 259–265. IEEE, 2017.

[55] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 921–937. IEEE, 2017.

[56] Hanqi Zhang, Xi Xiao, Francesco Mercaldo, Shiguang Ni, Fabio Martinelli, and Arun Kumar Sangaiah. Classification of ransomware families with machine learning based on n-gram of opcodes. *Future Generation Computer Systems*, 90:211–221, 2019.

ABDULRAHMAN ABU ELKHAIL (Member, IEEE) is a Ph.D. student in the Electrical and Computer Engineering Department at the University of Michigan, Dearborn. He has a B.S. in Computer Engineering from Yarmouk University, Irbid, Jordan and an M.S. in Computer Engineering from King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia. Before commencing his Ph.D., he worked in the industry for five years. His areas of interest include system security, automotive security, WSN and Ad Hoc networks, and IoT. He has several publications in refereed journals and conferences, as well as two U.S. patents.

NADA LACHTAR (Member, IEEE) received a bachelor's degree in network engineering and security from Jordan University of Science and Technology and the master's degree in computer and information science from the University of Michigan, Dearborn, where she is currently pursuing a Ph.D. degree in the same discipline. Her research interests include malware detection, system security, data privacy, and applied machine learning.

DUHA IBDAH (Member, IEEE) received a bachelor's degree in network engineering and security from Jordan University of Science and Technology and the master's degree in computer and information science from the University of Michigan, Dearborn, where she is currently pursuing a Ph.D. degree in the same discipline. Prior to joining the University of Michigan, she worked as a network engineer at Cisco Systems. Her research interests include systems security, networking, and privacy.

RUSTAM ASLAM is an undergraduate student in the College of Engineering and Computer Science at the University of Michigan, Dearborn. He is currently pursuing a B.S. in Computer Engineering. He has a strong passion for working with the latest technologies and is interested in specializing in the field of cybersecurity.

HAMZA KHAN is an undergraduate student at the University of Michigan. He is currently pursuing a B.S. in Computer Science and is in the third year of his degree. His areas of interest include security, data privacy, and applied machine learning.

ANYS BACHA (Member, IEEE) is an Assistant Professor at the University of Michigan. He leads the Security and Systems Lab which focuses on advancing the state-of-the-art in mobile and computer systems to address important challenges in security, applied machine learning, and energy efficiency. His research contributions have been published in top tier venues where his work received various prestigious awards. Furthermore, his industry impact is demonstrated through several U.S. and World patents. Prior to joining academia, he spent over 13 years in the industry where he worked in different Research and Development roles on a variety of subsystems spanning the hardware, firmware, and operating systems layers. He led multiple interdisciplinary efforts that include driving architectural changes into next generation Intel processors that are necessary to meet the demands of emerging workloads. During his tenure at Hewlett-Packard, Dr. Bacha led a group of engineers on a multi-million dollar scalable computing project that broke world records in performance in 2015 and 2014.

HAFIZ MALIK (Senior Member, IEEE) is currently a full Professor with the Electrical and Computer Engineering (ECE) Department, University of Michigan-Dearborn. He has published more than 100 articles in leading journals, conferences, and workshops. His research interests include automotive cybersecurity, IoT security, sensor security, multimedia forensics, steganography/steganalysis, information hiding, pattern recognition, and information fusion. Dr. is funded by the National Science Foundation, National Academies, Ford Motor Company, and other agencies. Since 2015, he has been a member of the MCity Working Group on Cybersecurity. He is a Founding Member of the Cybersecurity Center for Research, Education, and Outreach at UM-Dearborn. He is a Member Leadership Circle of the Dearborn Artificial Intelligence Research Center, UM-Dearborn. He is also a member of the Scientific and Industrial Advisory Board (SIAB), National Center of Cyber Security Pakistan.

APPENDIX

Category	Applications		Samples
Communication	*Beeper, Discord, Facebock Messenger, Flock, Geary, Google chats, ICQ, Kontact, Konversation, Zulip, MailSpring, Mattercord, Microsoft Team, 4K Video Downloader, NitroShare, Pidgin, Pixbuf, Signal, Skype, Slack, Telegram, Whatsdesk, Yak Yak, Remote Desktop Viewer*		24
Education	*Genius Math Tool, Grace, Klavaro, GLogic, Dictionary*		5
Entertainment	*Amarok, Angry Birds, Collision, Clementine Music Player, Google Play Music Player, SuperTuxKart, klines, Kodi, kblocks, Kubrick, kapman, MPV Media Player, Music Radar, Moviesquare, Musiko, Chess, Open Surge, Oh My Giraffe, Pin-Town, PlayOnLinux, Ri-li,Quadrapassel, Spotify, Squarehead, Lagno, Tiled Map Editor, VLC Media Player, Swell Foop, Tetravex*		29
Health & Fitness	*MyFitnessPal, WebMD, LiveStrong*		3
News	*Akregator, Liferea, QuiteRSS*		3
Productivity	*Android Studio, Atril Document Viewer, Audacity, Bitwarden, Boxy SVG, Brackets, Chartgeany, Tusk, Caffeine Indicator, Weather, Calc (Excel), Draw (Visio), Impress (Power Point), Writer (Word), Eclipse, Calibre, Carnet, Cheese, ClamTk, Converseen, Task Coach, Cura, Dropbox, Ebook-Viewer, WordPress, Filezilla, FocusWriter, Font Manager, Geany, Gifex, GitKraken, GLabels, GNOME Tweaks, Wakeup, Gscan2pdf, Handbrake, IDEA Ultimate, Inkscape, Gnome-Screenshot, Kdenlive Video Editor, Kontact, KolourPaint, Krita, MATE Dictionary, MusicBrainz Picard, MyPaint, Nomacs, OBS studio, Okular, Omniawrite, Organize My Files, PDF, PDF2GO, Peek, Plume Creator, Qalculate, QtQR, Rambox, Rainbow Board, Record My Desktop, Scribus, Shutter, Planner, SimpleScreenRecorder, SpeedCrunch, Speedy Duplicate Finder, Sublime Text, Subtitld, Synaptic Package Manager, Terminator, Calendar, TeXstudio, Thunderbird, TurboWrap, Ubuntu Cleaner, Vim, VirtualBox, Zotero-snap, CopyQ, Everpad, Kazam, Todoist, Texi2pdf, Visual Studio Code*		84
Social	*Corebird (twitter), Mumble, Somiibo*		3
Travel & Local	*Airbnb, Google Maps, TripAdvisor, Uber, Yelb*		5
Web Browser	*Arora, Chromium Web Browser, Firefox, Google Chrome*		4
Compression/Encryption	*pigz, xz, 7-zip, gzip, zip, lzop, lzma, pxz, bzip2, zstd, pax, tar, ar, Winzip, WinRAR,peazip, shar, cpio, iso, kgb, zpac, file roller, Lz4, plzip, pbzip2, p7zip, lbzip2, pixz,zpaq, zipx, AESCrypt, RSA*		32
Photo/Video Editing	*Kdenlive, PiTiVi, OBS Studio, Shotcut, OpenShot, Cinelerra, GIMP, DIGIKAM, Aoobe Photoshop, Fotoxx, Pinta, ShowFoto, AfterShot Pro, Photivo, UFRaw, Pixeluvo, RawTherapee*		17
Benchmark Suites	PARSEC 3.0	*Blackscholes, Bodytrack, Canneal, Dedup, Facesim, Ferret, Fluidanimate, Freqmine, Raytrace, Streamcluster, Swaptions, Vips, X264*	42
	SPLASH-3	*Barnes, Cholesky, Fft, Fmm, Lu, Ocean, Radiosity, Radix, Volrend, Water-nsquared, Water-spatial*	
	SPEC2K6	*Astar, Bwaves, Bzip2, CactusADM, Calculix, Gamess, Gcc, GemsFDTD, Gobmk, Gromacs, H264ref, Hmmer, Lbm, Leslie3d, Libquantum, Mcf, Milc, Namd, Omnetpp, Perlbench, Povray, Sjeng, Soplex, Sphinx3, Tonto, Wrf, Xalancbmk, Zeusmp*	

TABLE 7: Summary of benign applications and their corresponding categories.