

ShieldFS: The Last Word In Ransomware Resilient Filesystems

ANDREA CONTINELLA, Politecnico di Milano
ALESSANDRO GUAGNELLI, Politecnico di Milano
GIOVANNI ZINGARO, Politecnico di Milano
GIULIO DE PASQUALE, Politecnico di Milano
ALESSANDRO BARENGHI, Politecnico di Milano
STEFANO ZANERO, Politecnico di Milano
FEDERICO MAGGI, Politecnico di Milano and Trend Micro Inc.

Preventive and reactive security measures can only partially mitigate the damage caused by modern ransomware attacks. The remarkable amount of illicit profit and the cybercriminals' increasing interest in ransomware schemes demonstrate that current defense solutions are failing, and a large number of users are actually paying the ransoms. In fact, pure-detection approaches (e.g., based on analysis sandboxes or pipelines) are not sufficient, because, when luck allows a sample to be isolated and analyzed, it is already too late for several users! Moreover, modern ransomware implements several techniques to prevent detection by common AV. Similarly, for performance reasons, backups leave a small-but-important window of recent files unprotected.

We believe that a forward-looking solution is to equip modern operating systems with generic, practical self-healing capabilities against this serious threat.

We present SHIELDFS, a drop-in driver that makes the Windows native filesystem immune to ransomware attacks, even when detection fails SHIELDFS dynamically toggles a protection layer that acts as a copy-on-write mechanism whenever its detection component reveals suspicious activity. For this, SHIELDFS monitors the filesystem's internals to update a set of adaptive models that profile the system activity over time. This detection is based on a study of the filesystem activity of over 2,245 applications, and takes into account the entropy of write operations, frequency of read, write, and folder-listing operations, fraction of files renamed, and the file-type usage statistics. Additionally, SHIELDFS monitors the memory pages of each "potentially malicious" process, searching for traces of the typical block cipher key schedules.

We show how SHIELDFS can shadow the write operations. Whenever one or more processes violate our detection component, their operations are deemed malicious and the side effects on the filesystem are transparently rolled back.

We demonstrate how effective SHIELDFS is against samples from state of the art ransomware families, showing that it is able to detect the malicious activity at runtime and transparently recover all the original files.

1 INTRODUCTION

Ransomware [20] is a class of malware that encrypts valuable files found on the victim's machine and asks for a ransom to release the decryption key(s) needed to recover the plaintext files. The requested ransom payment is typically in the order of a few hundreds US dollars [15] (or equivalent in crypto or otherwise untraceable currency [17]). Clearly, the success of these attacks depends on whether most of the victims agree to pay (e.g., because of the fear of losing their data). Unfortunately, according to a thorough survey dated November 2015 [4], about 50 percent of ransomware victims had surrendered to the extortion scheme, resulting in million of dollars of illicit revenue. In March 2014, Symantec estimated that the Cryptowall gang has earned more than \$34,000 in its first month of activity. In June 2015, the FBI's Internet Crime Complaint Center [5] reportedly received 992 Cryptowall-related complaints between April 2014 and June 2015, totaling \$18M worth of losses.

In the first three months of 2016, according to a recent analysis [13], more than \$209 million in ransomware payments were made in the US alone. From a technical viewpoint, ransomware families are now quite advanced. While first-generation ransomware were cryptographically weak, the recent families encrypt each file with a unique symmetric key protected by public-key cryptography. Consequently, the chances of a successful recovery (without paying the ransom) have drastically decreased [1, 10].

Problem Statement and Vision. Kharraz et al. [9] were the first to analyze a large corpus of ransomware samples. The authors suggest that the filesystem is a strategic point for monitoring the typical ransomware activity. In this paper, we set the next research objective: Creating a forward-looking filesystem that transparently prevents the effects of ransomware attacks on the data. We make a step toward such vision by proposing, implementing and evaluating an approach that combines automatic detection and transparent file-recovery capabilities at the filesystem level, all combined in a ready-to-use Windows driver.

Preliminary Feasibility Assessment. Our first goal is to understand how ransomware *compares* to benign software from the filesystem’s viewpoint. We start by analyzing in-depth how benign software typically interacts with the filesystem on *real-world* computers. We use the I/O request packets (IRPs) as the focal point of our analysis, as IRPs are the basic data units originating from high-level operations (e.g., read file, open file). In practice, we performed the first large-scale data collection of IRPs from real-world, ransomware-free machines, to profile the low-level filesystem activity in normal conditions. To this end, we developed IRPLogger, a data-collection agent that we installed on 11 machines used by volunteers for their typical day-to-day tasks (i.e., personal, office, and development). We anonymized and collected about a month worth of data, gathering more than 1.7 billion IRPs generated by 2,245 distinct applications (we will make this data available to other researchers). Using this collected data as a reference, we populated a set of analysis machines with files and directory trees such that they resemble the typical filesystem organization and content observed in the 11 real-world machines. This step is essential to create a realistic environment such that to trigger the ransomware attacks. We then used IRPLogger to monitor the filesystem on such machines infected by state of the art ransomware samples.

Proposed Approach. Our preliminary assessment guided us to design a detection system based on the combined analysis of entropy of write operations, frequency of read, write, and folder-listing operations, dispersion of per-file writes, fraction of files renamed, and the file-type usage statistics. Our approach is to automatically create detection models that distinguish ransomware from benign processes at runtime SHIELDIFS adapts these models to the filesystem usage habits observed on the protected system. Additionally, SHIELDIFS looks for indicators of the use of cryptographic primitives. In particular, SHIELDIFS scans the memory of any process considered as “potentially malicious,” searching for traces of the typical block cipher key schedules.

A distinctive aspect of SHIELDIFS is how it copes with code injection, a common technique used by modern ransomware (as well as other malware). With code injection, a perfectly legitimate process suddenly executes malicious code. Our detection mechanism takes into account both the long- and the short-term history of each process, and of the entire system. Indeed, we are agnostic with respect to how the infection has bootstrapped (e.g., malicious executable, remote code execution) and on the availability of the executable. Rather, we focus on the runtime *effects* on the target system. In fact, as observed in [19], the activity of modern malware can span across multiple process and OS facilities, and, more importantly, an isolated sample to analyze is a luxury in early stage of spreading campaigns. Therefore, detection systems should not assume that a binary executable is available.

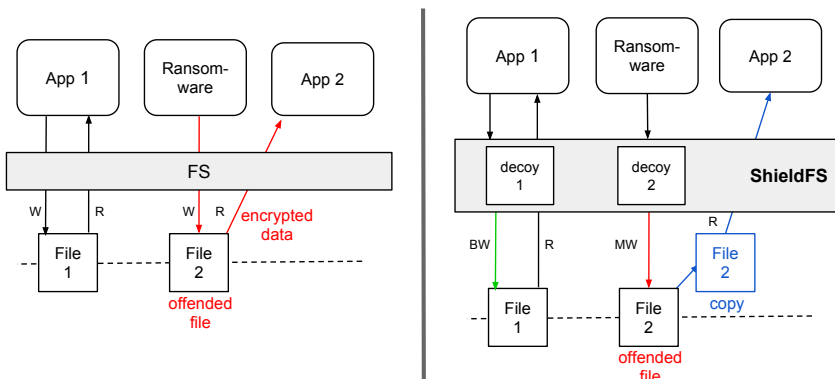


Fig. 1. On the right SHIELDDFS shadowing a file offended by ransomware malicious write (MW), in comparison to standard filesystems (on the left).

We apply our detection approach in a real-time, self-healing virtual filesystem that shadows the write operations. Thus, if a file is surreptitiously altered by one or more malicious processes, the filesystem presents the original, mirrored copy to the user space applications. This shadowing mechanism is dynamically activated and deactivated depending on the outcome of the aforementioned detection logic. Figure 1 depicts the logical activity of SHIELDDFS in comparison with a traditional filesystem.

Experimental Results Summary. We evaluated SHIELDDFS on 688 samples from 11 distinct families, showing that it can successfully protect user data from real-world attacks performed by recent, state-of-the-art malware families. The system exhibited remarkable accuracy and generalization capabilities even when evaluated via cross-validation on the large dataset that we collected from the 11 real-world machines. Also, we installed SHIELDDFS on the personal machines in use by 3 volunteers, on which it correctly identified ransomware processes, and successfully reverted their effects. The performance impact of our prototype implementation is such that SHIELDDFS is applicable in real-world settings.

Summary of Original Contributions.

- We performed the first, large-scale data collection of I/O request packets generated by *benign* applications in real-world conditions. Our dataset contains about 1.7 billion IRPs produced by 2,245 different applications.
- We propose a ransomware-detection approach that enables a modern operating system to recognizing the typical signs of ransomware behaviors.
- We propose an approach that makes a modern filesystem resilient to malicious encryption, by dynamically reverting the effects of ransomware attacks.
- We implemented these approaches as a drop-in, Windows kernel module that we showed capable of successfully protecting from current ransomware attacks.

2 LOW-LEVEL I/O DATA COLLECTION

To understand how ransomware typically interact with the filesystem in comparison to benign applications, the main challenge is to be able to observe them in their usual working conditions (e.g., on a victim’s machine). Since there is no such recent data for this purpose, we collected it from real, operational desktop computers for several weeks. First, this provided us with a real-world reference “picture” of how files and folders are organized in a typical computer, which is useful to reproduce an environment that triggers the ransomware activity. Secondly, this approach provided

Table 1. Statistics of the collected low-level I/O data from 11 real-world machines.

User	Win. ver.	Usage	Data [GB]	#IRPs Mln.	#Procs Mln.	Apps	Period [hrs]	Data Rate [MB/min]
1	10	dev	3.4	230.8	16.60	317	34	7.85
2	8.1	home	2.4	132.1	9.67	132	87	2.04
3	10	office	0.9	54.2	5.56	225	17	0.83
4	7	home	4.7	279.9	18.70	255	122	5.18
5	7	home	2.2	138.1	5.04	141	47	4.10
6	10	dev	1.8	100.4	10.30	225	35	2.42
7	8.1	dev	0.8	49.0	3.28	166	8	5.62
8	8.1	home	0.8	43.9	6.33	148	32	2.16
9	8.1	home	7.7	501.8	24.20	314	215	3.21
10	7	home	0.9	57.6	2.63	151	18	4.60
11	7	office	2.6	175.2	4.69	171	28	8.51
<i>Total</i>			28.2	1,763.0	107.00	2245	643	-

us with a large dataset of filesystem access patterns originating from *benign* applications while exercised by real-user interactions. This is essential to verify whether ransomware and benign applications interact with the filesystem in a significantly different way that could be leveraged for detection.

To carry out our analysis, we developed IRPLogger, a low-level I/O filesystem sniffer, which we installed on real-world machines in use by 11 volunteers. We can categorize the participants as “home,” “developer,” or “office” users. As summarized in Table 1, we collected 28.2 GB of compressed and anonymized data, corresponding to 1,763 million IRPs.

2.1 Filesystem Sniffer Details

At the first boot, IRPLogger traverses the directory tree of each mounted drive to collect metadata including total number of files, number of files per extension, and directory depth. The core of IRPLogger is a minifilter driver [7] that intercepts the I/O requests generated for each filesystem primitive invoked by userland code (e.g., `CreateFile`, `WriteFile`, `ReadFile`). IRPLogger enriches the raw IRPs with data including timestamp, writes entropy, and PID. An example log entry (before anonymization) is as follows:

```
<time, program name, PID, IRP op, entropy, file info>
```

When run on the participants’ machines, IRPLogger minimizes and hashes any privacy-sensitive data such as the file names and paths. We keep the extension of the accessed files in clear, as this detail is needed for computing per-type file statistics and features. Before collection, the logs are split into sessions and compressed for space efficiency.

Table 2. Statistics of the collected low-level I/O data from 383 ransomware samples.

Ransomware Family	No. Samples	Data	#IRPs Millions
CryptoWall	157 (41.0%)	8.0	286.7
Crowti	125 (32.6%)	5.7	173.1
CryptoDefense	77 (20.1%)	4.5	171.6
Critroni	14 (3.7%)	0.6	3.0
TeslaCrypt	10 (2.6%)	0.9	29.2
<i>Total</i>	383	19.7	663.6

2.2 Ransomware Activity Data Collection

We leveraged IRPLogger also to collect ransomware activity data. During December 2015 we used the VirusTotal Intelligence API to obtain the most recent Windows executables consistently labeled with the main ransomware families (i.e., CryptoWall, TeslaCrypt, Critroni, CryptoDefense, Crowti). We manually ran each sample to ensure that it was fully and properly working (e.g., some samples did not receive instructions and public encryption keys from the attacker’s control servers), so obtaining the 383 active ransomware samples summarized in Table 2.

Then, we prepared a set of virtual machines on which we activated IRPLogger running on top of Windows 7 (64-bit). We installed common utilities such as Adobe Reader, Microsoft Office, alternative Web browsers, and media players. To create a legitimate-looking system, we included typical user data such as saved credentials, browser history, and realistic decoy files (e.g., images, documents), such that to trigger the samples. We used real files—collected by randomly crawling web search-engines results—reflecting file-type and directory tree distribution of the aforementioned 11 clean machines. At runtime, our analysis environment emulates basic user activity (e.g., moving the mouse, launching applications). Following the best practices for malware experiments suggested by [14], (1) we let the malware executables run for 90 minutes, (2) we allowed the samples to communicate with their control servers, and (3) denied any potentially harmful traffic (e.g., spam) during the experiments. For the sake of scientific repeatability, we are open to provide access to (or the implementation details of) our analysis environment. After each execution, we saved the IRP logs and rolled back each virtual machine to a clean snapshot.

2.3 Filesystem Activity Comparison

The remarkable differences in the features distribution shown in Table 3 confirms ransomware and benign applications are different filesystem-wise, and motivates us to exploit these results to create a full-fledged remediation system.

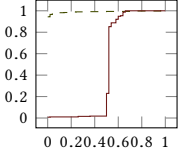
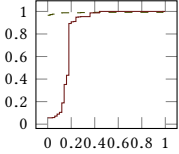
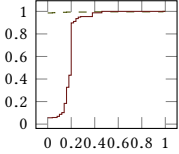
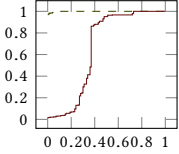
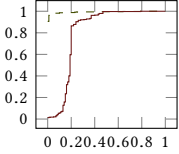
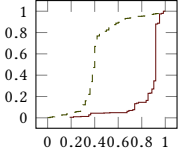
We focus our analysis on user data, that is, the main target of ransomware attacks. Contrarily, benign programs, especially system processes (e.g., services, updates manager), access large portions of files in dedicated folders, or in the system folders. For this reason, we separate the IRP logs of user folders from the IRPs of system folders. In practice, we compute the features listed in Table 3 twice: first on IRP logs of user paths only (e.g., excluding WINDOWS or Program Files), and then on *all* paths.

3 APPROACH AND METHODOLOGY

For clarity, we logically divide our approach into two parts: ransomware activity detection and file recovery. Our file-recovery approach is inspired by copy-on-write filesystems and consists in automatically shadowing a file whenever the original one is modified, as depicted in Figure 1. Benign modifications are then asynchronously cleared for space efficiency, and the net effect is that the user never sees the effects of a malicious file encryption.

We consider all files as “decoys,” that is, we assume that the malware will reveal its behavior because, indeed, it cannot avoid to access the files that it must encrypts to fulfill its goal. The features defined in Table 3 summarize the I/O-level activity recorded on these decoys into quantitative indicators of compromise. Thus, the detection and file-recovery parts of our approach are tightly coupled, in the sense that we rely on such decoys to both (1) collect data for detection, and (2) manage the shadowing of the original files.

Table 3. We use these features for both our preliminary assessment (Section 2) and as the building block of the SHIELDIFS detector (Sections 3 4). SHIELDIFS computes each feature multiple times while monitoring each process, on various portions of filesystem activity, as explained in details in Section 3.1. We normalize the feature values according to statistics of the file system (e.g., total number of files, total number of folders). This normalization is useful to adapt SHIELDIFS to different scenarios and usage habits. The rightmost column shows a comparison of benign (---) vs. ransomware (—) programs by means of the empirical cumulative distribution, calculated on the datasets summarized in Table 1 and 2, respectively. We notice that ransomware activity is significantly different than that of benign programs according to our features, suggesting that there is sufficient statistical power to tell the two types of programs apart.

Feature	Description	Rationale	Comparison
#Folder-listing	Number of folder-listing operations normalized by the total number of folders in the system.	Ransomware programs greedily traverse the filesystem looking for target files. Although filesystem scanners may exhibit this behavior, we recall that ransomware programs will likely violate multiple of these features in order to work efficiently.	
#Files-Read	Number of files read, normalized by the total number of files.	Ransomware processes must read all files before encrypting them.	
#Files-Written	Number of files written, normalized by the total number of files in the system.	Ransomware programs typically execute more writes than benign programs do under the same working conditions.	
#Files-Renamed	Number of files renamed or moved, normalized by the total number of files in the system.	Ransomware programs often rename files appending a random extension during encryption.	
File type coverage	Total number of files accessed, normalized by the total number of files having the same extensions.	Ransomware targets a specific set of extensions and strives to access all files with those extensions. Instead, benign application typically access a fraction of the extensions in a given time interval.	
Write-Entropy	Average entropy of file-write operations.	Encryption generates high entropy data. Although file compressors are also characterized by high-entropy write-operations, we show that the combined use of all these features will mitigate such false positives. Moreover, we notice that our dataset of benign applications contains instances of file-compression utilities.	

3.1 Ransomware FS Activity Detection

Given the results of our preliminary data analysis in Section 2.3, and the aforementioned assumptions and design decisions, we approach the detection problem as a supervised classification task. Specifically, we propose a custom classifier trained on the filesystem activity features defined in Table 3, extracted from a large corpus of IRP logs obtained from clean and infected machines. Once trained, this classifier is leveraged at runtime to decide whether the features extracted from a live system fit the learned feature distributions (i.e., no signs of malicious activity) or not.

Process- and System-centric FS Models. A malware can perform all its malicious actions on a single process, or split it across multiple processes (for higher efficiency and lower accountability). For this reason, our custom classifier adopts several models. One set of models, called *process centric*, each trained on the processes individually. A second model, called *system centric*, trained by considering all the IRP logs as coming from a single, large “process” (i.e., the whole system). The rationale is that the system-centric model has a good recall for multi-process malware, but has potentially more false positives. For this reason, the system-centric model is used only in combination to the process-centric model.

Incremental, Multi-tier Classification. Although our file-recovery mechanism is conservative, we want to minimize the time to decision. Moreover, since the decision can change over time, all processes must be frequently and efficiently monitored. To obtain an acceptable trade off between speed and classification errors we adopt two orthogonal approaches.

First, (1) instead of running our classifiers on the entire available process data, we split the data in intervals, or *ticks*. Ticks are defined by the fraction of files accessed by the monitored process—with respect to the total number of files in the system. In this way, we obtain an array of incremental, “specialized” classifiers, each one trained on increasingly larger data intervals. For instance, when a process has accessed 2% of the files, we query the “2%-classifier” only, and so on. Our experiments (Figure 5) show that this technique reduces the #IRPs required to cast a correct detection by three orders of magnitude, with a negligible impact on the accuracy.

Secondly, (2) to account for changes during a process’ lifetime, we monitor both the long- and short-term history. In practice, we organize the aforementioned incremental classifiers in a multi-tier, hierarchical structure (as depicted in Figure 2), with each tier observing larger spans of data. At each tick, each tier analyzes the data up to N ticks in the past, where N depends on the tier level. We label a process as “ransomware” as soon as at least one of tiers agrees on the same outcome for K consecutive ticks. In Section 5 we show that the choice of K has negligible impact on the false positives.

Example (Code Injection). This example explains how our incremental, multi-tier models handle a typical case. A benign process (e.g., Explorer) is running, and has accessed some files. For the first i ticks SHIELD FS will classify it as benign. Now, the Ransomware process injects its code into Explorer’s code region. Referring to Figure 2, if Ransomware does code injection after the 3rd tick, the global-tier model classifies Explorer as benign, because the long-term feature values are not be affected significantly by the small, recent changes in the filesystem activity of Explorer. Instead, the tier-1 model identifies Explorer as malicious, because the tier-1 features are based *only* on the most recent IRPs (i.e., those occurring right after the code injection). The same applies for tier-2 models after the 4th tick, and so on. If $K = 3$, for instance, and all the triggered tiers agree on a positive detection, the Explorer process is classified as malicious at this point in time. This decision, clearly, can change while more process history is examined.

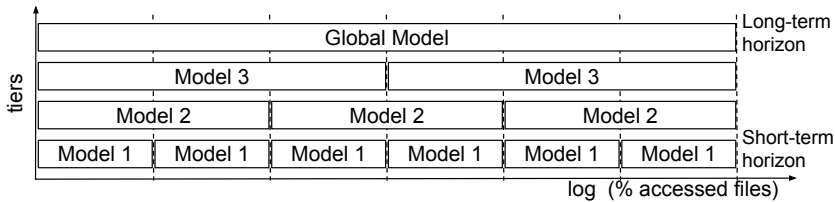


Fig. 2. Example of the use of incremental models. At each interval, we check simultaneously multiple incremental models at all applicable tiers.

3.2 Cryptographic Primitives Detection

Detecting traces of a cipher within a suspicious process memory, in addition to malicious filesystem activity, is a further indication of its ransomware nature. The malware authors’ goal is to efficiently encrypt large sets of files, using a single master key per victim. Thus, instead of relying directly on asymmetric cryptography, which is resource intensive, the strategy is to encrypt each file with a symmetric cipher and a per-file random key, each encrypted with an asymmetric master secret obtained from the attacker’s control server.

Efficient Block Ciphers. The most widespread, efficient symmetric-encryption algorithms of choice are fast block ciphers. These ciphers combine the plaintext with a secret key through a sequence of iterations, known as *rounds*. In particular, the key is expanded in a sequence of values, known as the *key schedule*, which is employed to provide enough key material for the combination during all the rounds. Since the key expansion is deterministic and depends on the key alone, it can be pre-computed and reused, with a significant performance gain (e.g., 2 to 4× in case of AES-128). Indeed, all the mainstream cryptographic libraries (e.g., OpenSSL, mBED TLS) and the vast majority of ransomware families do pre-compute the key schedule.

Side Effects. The main side effect of such a pre-computation technique is that the entire key schedule is (and must remain) materialized in memory during all the encryption procedure. We leverage this side effect, and perform a scan of the memory of the running process, checking, at every offset, whether the content of the memory can be obtained as a result of a key schedule computation. Due to the tight constraints present between the key and the expanded key (i.e., sound key schedules impose a bijection between them) it is extremely unlikely that a random sequence of bytes accidentally matches the result of a key expansion, making false positives very unlikely. False negatives may occur if the key schedule is not contiguously stored in memory. However, due to the small size of the involved data (i.e., less than a single 4kiB page), such an event is unlikely to happen due to memory allocation fragmentation.

Note. Although this technique has the benefit of recovering the secret keys used during the encryption, relying exclusively on this criterion for file recovery would not be generic and future-proof: Since each file may be encrypted with a dedicated symmetric key, to guarantee the recoverability of all files, the memory scanning action should be continuous, and there is the risk that some keys are simply missed. Instead, by using our dual approach (i.e., filesystem and memory analysis) SHIELDIFS can guarantee the recoverability of all files, regardless of how they are encrypted.

3.3 Automatic File Recovery Workflow

When SHIELDIFS is active, any newly created process enters a so-called “unknown” state. Whenever such a process opens a file handle in write or delete mode *for the first time* (only), SHIELDIFS copies the file content in a trusted, read-only storage area. This storage can be on the main drive or on a secondary drive. In either case, SHIELDIFS denies access to this area from any userland process by discarding any modification request coming from the upper I/O manager. From this moment on,

the process may read or write such file, while SHIELDFS monitors its activity. When SHIELDFS has collected enough IRPs, the process goes into a “benign,” “suspicious,” or “malicious” state.

File copies belonging to “benign” processes can be deleted immediately or, as SHIELDFS does, scheduled for asynchronous deletion. Since storage space is convenient nowadays, leaving copies available for an arbitrarily long time delay does not impose high costs. In turns, it greatly benefits the overall system performance because, by acting as a cache, it limits the number of copy operations required when the same files are accessed (and would need to be copied) multiple times.

For any process that enters the “malicious” state for at least one tick, SHIELDFS checks the presence of ciphers within the process. If any are found, it *immediately* suspends the process and restores the offended files. Otherwise, it waits until K positive ticks are reached before suspending the process, regardless of whether a traces of ciphers are found.

Processes can enter a “suspicious” state when the process-centric classifier is not able to cast a decision. In this case, SHIELDFS queries the system-centric model. If it gives a positive outcome, then the process enters the “malicious” state. Otherwise the process is classified as “benign.”

4 SHIELDFS SYSTEM DETAILS

We implemented SHIELDFS following the high-level architecture depicted in Figure 3, and the detection loop defined in Algorithm 1. We focused on Microsoft Windows because it is the main target of the vast majority of ransomware families. We argue that the technical implementation details may change depending on the target filesystem and OS’s internals. However, our approach does not require any special filesystem nor OS support. Thus, we expect that it could be ported to other platforms with modest engineering work.

4.1 Ransomware FS Activity Detection

To intercept the IRPs, SHIELDFS registers callback functions through the filter manager APIs (i.e., `FltRegisterFilter`). For each IRP, the called function updates the feature values, using separate kernel worker threads for computation-intensive functions (e.g., entropy calculation).

Feature Normalization. To keep the feature values normalized (e.g., number of files read, normalized by the total number of files), the first time the SHIELDFS service is run, it scans the filesystem to collect the file extensions, number of files per extensions, and overall number of files.

Since the normalization factors change over time (i.e., new, deleted, or renamed files), SHIELDFS updates them in two ways. One mechanism uses a dedicated kernel thread to update the normalization factors *in real time*. This has no performance impact, since SHIELDFS already keeps track of the relevant file operations. However, an attacker could exploit it to bias the feature values, by manipulating the normalization factors (e.g., by creating many legitimate, low-entropy files). The second mechanism raises the bar for the attacker because it updates the normalization factors *periodically* (e.g., once a day). In this way, even if an attacker tries to manipulate our normalization factors, she will need to wait until the next update before starting to access files without triggering any of the features. Although the second mechanism is more resilient to such attacks, it is prone to false positives if users create many files between updates. False positives, however, occur only if a significant number of files are accessed in a way that resembles a ransomware activity (i.e., several folder-listing operations, followed by file reads or renaming, and high-entropy writes). Taking our dataset of benign machines monitored for about a month as a reference, the impact of these false positives is very low compared to the benefits of increased resiliency.

Classifier Details. Each classifier is implemented as a random forest of 100 trees. Each tree outputs either -1 (benign) or $+1$ (malicious). The overall outcome of each process-centric classifier is the sum of its trees’ outcome, from -100 (highly benign) to 100 (highly malicious). In case of a tie

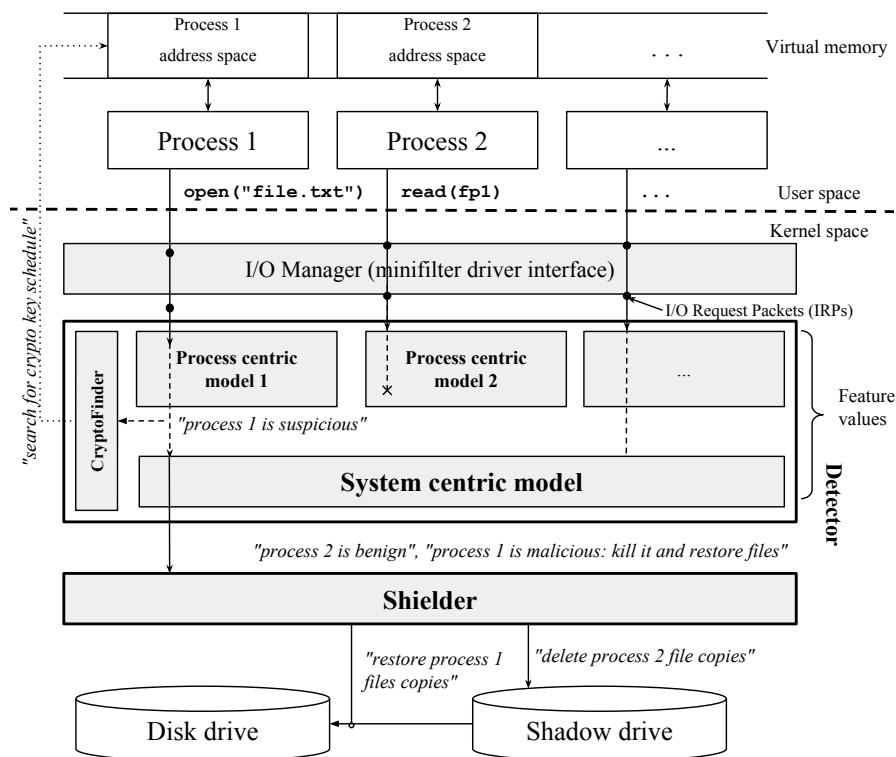


Fig. 3. High-level overview of SHIELDFS. The Detector and the Shielder are Windows minifilter drivers, and the CryptoFinder is kernel driver.

(i.e., zero), SHIELDFS marks the monitored process as “suspicious,” and invokes the system-centric classifier to take the decision. In case of a second tie, we conservatively consider the process as malicious.

Monitoring Ticks. SHIELDFS gives more relevance to small variations in a feature value when a process has only accessed a few files. At the same time it minimizes the total number of models needed, so as to contain the performance impact. For these reasons, the size of each tick grows exponentially with the percentage of files accessed by a process. After careful evaluation, we used 28 tiers, for intervals ranging from 0.1 to 100%, each one corresponding to a distinct model tier. Adding other ticks beyond 28 would yield no improvements in detection rates, and would instead penalize the performance.

Countermeasure for Buffer-file Abuse. Some versions of Critroni exploit one single file as a write-and-encrypt-buffer. Specifically, the malware moves the target original file in a temporary file, encrypts it, and then overwrites the original file with it. As a result, SHIELDFS observes many renaming operations, followed by many read-write operations on a single file, thus biasing the feature values.

To counteract this evasion technique, SHIELDFS keeps track of when a file is read (or written) right after being renamed (or moved), such that to update the feature values taking into account the net, end-to-end effect, as if the buffer file was not used. This mechanism comes at no extra cost, since SHIELDFS already keeps track of file-renaming operations.

Algorithm 1 Detection routine for each process.

```
1: procedure ISRANSOMWARE( $PID, fs\_features$ )
2:    $crypto \leftarrow \perp$ 
3:   for  $tier \in \{1, \dots, top\}$  do
4:     if  $enoughFilesAccessedForTickOf(tier)$  then
5:        $result \leftarrow ProcessClassifier_{tier}(fs\_features)$ 
6:        $resetFeatureValues(tier)$ 
7:       if  $result < 0$  then
8:          $K_{tier} \leftarrow 0$ 
9:       else
10:         $crypto \leftarrow CryptoFinder(PID)$ 
11:        if  $result = 0$  then
12:          if  $SystemClassifier_{tier}(fs\_features) \geq 0$  then
13:             $K_{tier} ++$ 
14:          else
15:             $K_{tier} ++$ 
16:        if  $crypto \vee \exists tier : K_{tier} \geq K_{threshold}$  then
17:
18:          return malicious
19:
20:        return benign
```

4.2 Cryptographic Primitives Detection

SHIELDIFS checks the memory of processes classified as “suspicious” or “malicious” for the presence of symmetric cryptographic primitives. For the sake of clarity, we remark that the output of `CryptoFinder` is used as an additional, non-essential feature. Hence, SHIELDIFS is able to detect even samples that do not show any encryption process, as long as the filesystem activity models are sufficiently (i.e., at least K positive ticks) triggered.

SHIELDIFS does *not* make any assumption on *how* the cipher is implemented by the malware, save for the materialization of the key schedule. As a proof of concept, we select AES as our target block cipher, due to its widespread use. AES’s key schedule expands 128, 192 or 256 key bits into 1408, 1664 or 1920 key schedule bits, respectively. As a consequence, taking all the 2^{64} possible positions in the address space as candidates, and assuming that the accidental occurrence of a key expansion for a location is independent from it occurring for a different one, the probability of a false positive is $2^{64}2^{-1408} = 2^{-1344}$ (in the most favorable case), which is negligible for practical purposes.

`CryptoFinder` receives the PIDs of suspicious processes by the Detector, through IOCTL. When triggered, `CryptoFinder` attaches to a process and obtains the list of its memory pages. Specifically, `CryptoFinder` looks only at the *committed* pages, defined in Windows as the pages for which physical storage has been allocated—either in memory or in the paging file on disk. Then, `CryptoFinder` runs the key-schedule algorithm on these memory regions and checks whether its expansion occurs. For efficiency reasons, we stop the inspection of a location as soon as there is a single byte mismatch.

4.3 Automatic File Recovery

We implemented Shielder as a Windows minifilter driver that monitors file modifications by registering a callback for those `IRP_MJ_CREATE` operations which security context parameter `Parameters.Create.SecurityContext` indicates a “write” or “delete” I/O request. If the target file is not shadowed yet, SHIELDIFS creates a copy before letting the request through. With the same technique it monitors the destination of (potentially malicious) file-renaming operations,

by hooking the `IRP_MJ_SET_INFORMATION` requests having the `ReplaceIfExists` flag set. File handing and indexing in the shadow drive is based on the `FILE_ID` identifier assigned by NTFS to each file.

Transaction Log. SHIELDIFS maintains a transaction log of the relevant IRPs (e.g., those resulting from file modifications). Whenever a process is classified as malicious, SHIELDIFS inspects such log and restores each file affected by the offending process.

File copies are deleted only when the processes that modified the original file have been cleared as “benign.” SHIELDIFS treats the shadow drive as a cache: It avoids shadowing the same file if a fresh copy (i.e., not older than T hours) already exists. According to our experiments, based on the workload of real-world users (obtained from our large-scale data collection), the age T imposes acceptable overhead (below 1%) and can be safely set to any number between 1 and 4. In Section 6 we discuss how the choice of T raises the bar for the attacker who wants to successfully encrypt a large portion of files.

Whitelisting of Support Files. Files that have no value for a user are of no interest for ransomware attacks. An example are application-support directories, which contain cache or temporary files, which are frequently accessed by benign applications. These folders can be safely whitelisted to reduce the performance overhead due to the frequent operations on such files. To avoid that an attacker could exploit the whitelisted folders as a “demilitarized zone” where to copy the target files (prior to encrypting them), we adopt the following solution. Any process that has *never* accessed a whitelisted folder is considered “suspicious” as soon as it attempts to move files into it. The files offended by this operation are preemptively shadowed.

Windows Shadow Copy. Recent Windows versions have a so-called Volume Shadow Copy Service. However, Windows shadow copies have two issues. First, the copies are created only during the next power down and boot cycle. Instead, as we already mentioned, our approach is designed for short-term backup that can allow users to restore recently modified files. Secondly, shadow copies can be easily bypassed and deleted, as most of recent ransomware families do before starting the encryption process [10].

5 EXPERIMENTAL RESULTS

As we did for our preliminary analysis (Section 2.2), we evaluated SHIELDIFS on an analysis environment with virtual machines provisioned so as to mimic the file content and organization of potential victim machines.

We first performed a thorough *cross validation* to assess (1) the generalization capabilities of our classifiers, and (2) the impact of the parameter choice on the overall detection quality and performance. Second, we infected physical machines *in use by real users* (for their day-to-day activities) with 3 samples of ransomware families. SHIELDIFS was able to detect their activity and fully recover all the compromised files. Third, we evaluated the *detection and file-recovery capabilities* against ransomware samples that SHIELDIFS has never seen before. Last, we measured the performance overhead of SHIELDIFS by considering the typical usage workload, where “typical” refers to our initial large-scale collection of I/O filesystem logs.

A video demo of SHIELDIFS in action is available on YouTube at [2].

5.1 Detection Accuracy

Cross validation allows to reveal the presence of overfitting-induced biases and thus is a crucial aspect of any machine-learning-based approach. We conducted three cross-validation experiments to evaluate the quality of the Detector on our dataset of 383 ransomware samples and 2,245 benign applications from the 11 user machines. We count positive or negative detections at the

process granularity, and calculate the TPR and FPR based on the true overall number of benign and ransomware processes.

10-fold Cross Validation. We calculated the true- and false-positive rate on 10 random train/test splits. Figure 4 and 5 show the TPR and FPR in function of the minimum percentage of files, and #IRPs, respectively, needed to cast the decision. The results show the benefit of the system-centric model as a tie breaker, and the incremental approach as an early detector, which requires orders of magnitude less IRPs to cast a decision, with almost no impact on the FPR (i.e., from 0.0 to 0.00015 in the worst case).

One-machine-off Cross Validation. To further show the independence of our detection results from the specific machine that generates the benign subset of training and testing data, we performed a per-machine cross validation. We selectively removed the data of one machine from the training set, and used it as the testing set. We repeated this procedure for each of the 11 machines.

Table 4 shows (1) that SHIELDIFS has no strong dependency from the training-testing data split, and (2) confirms that the system-centric model is useful to reduce FPs by acting as a tie breaker.

Causes of False Positives. We found only two cases of false positives. For the first user machine, the detector triggered because explorer . exe biased the normalization, by accessing a very large number of files (more than the normalization factors, which were not up to date). This motivated us to implement the mechanism that live-updates the system-wide, feature counts for normalization (rather than doing such an update periodically). This *eliminates the false positives*, creating however a small opportunity for the attacker to bias the normalization factors. This trade off is clearly inherent in the statistical nature of SHIELDIFS.

Interestingly, in 4 out of 11 machines we found activity of the WinRAR file-compression utility, which performed high-entropy writes. Nevertheless, WinRAR was correctly classified as benign, thanks to the contribution of the remainder features.

The second false positive was Visual Studio, which wrote 175 files, with a very high average entropy (0.948). This was an isolated case, which happened only on one of the 32 Visual Studio sessions recorded in our dataset.

Parameter Setting. The choice of K , the number of consecutive positive detections required to consider a process as malicious, can be set to minimize the FPR to zero, at the price of a very small variation (within +/-0.5%) of TPR. Or vice versa. Table 5 shows that setting $K = 3$ maximizes the TPR, with very few false positives. Instead, with $K = 6$, SHIELDIFS did not identify a sample that performed injection into a benign process and that encrypted files very slowly. Generally,

Table 4. FPR with One-machine-off Cross Validation

User		False positive rate [%]		
Machine	Process	System	Outcome	
1	0.53	23.26		0.27
2	0.00	0.00		0.00
3	0.00	0.00		0.00
4	0.00	1.20		0.00
5	0.22	45.45		0.15
6	0.00	4.76		0.00
7	0.00	88.89		0.00
8	0.00	0.00		0.00
9	0.00	0.00		0.00
10	0.00	0.00		0.00
11	0.00	0.00		0.00

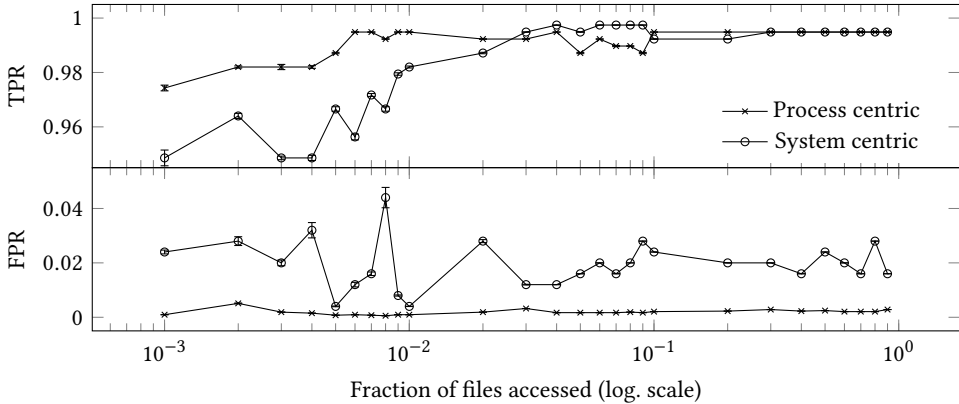


Fig. 4. 10-fold Cross Validation: Average and standard deviation of TPR and FPR with process- vs. system-centric detectors.

false negatives are more expensive than false positives in ransomware-detection problems, thus we advise for values of K that maximize the TPR. This has the additional benefit of reducing the number of IRPs required for a correct detection.

5.2 Protection of Production Machines.

In order to evaluate our system in real scenarios, we tested SHIELDFS on three distinct real machines (running Windows 7 and 10), in use by real users for their day-to-day activities for years, containing 2,319, 165,683, and 144,868 files, respectively. We randomly selected 3 samples¹ from our dataset (Critroni, TeslaCrypt, and ZeroLocker) and manually analyzed them to ensure that they were not stealing any personal information. After cloning the hard drives as a precaution, we installed SHIELDFS, and infected the machines. All the three samples were correctly detected and all the affected files were correctly restored automatically.

5.3 Detection and Recovery Capabilities

We setup an environment as described in Section 2.2, with dummy files to reproduce a real-user setting. Moreover, we stored 9,731 files typically targeted by ransomware attacks (e.g., images and documents of various formats), of which we pre-calculated the MD5 for integrity verification after each experiment. We then trained SHIELDFS on the large dataset of IRP logs collected as part of our preliminary analysis.

Dataset of Unseen Samples. In addition to the cross-validation experiments on 383 samples, which already show the predictive and generalization capabilities of SHIELDFS, we obtained 305 novel,

¹ e89f09fdded777ceba6412d55ce9d3bc, 209a288c68207d57e0ce6e60ebf60729, bd0a3c308a6d3372817a474b7c653097

Table 5. 10-fold Cross-Validation: Choice of K .

K	FPR	TPR	IRPs
1	0.208%	100%	35664
2	0.076%	100%	43822
3	0.038%	100%	67394
4	0.019%	99.74%	80782
5	0.019%	99.74%	104340
6	0.000%	99.74%	135324

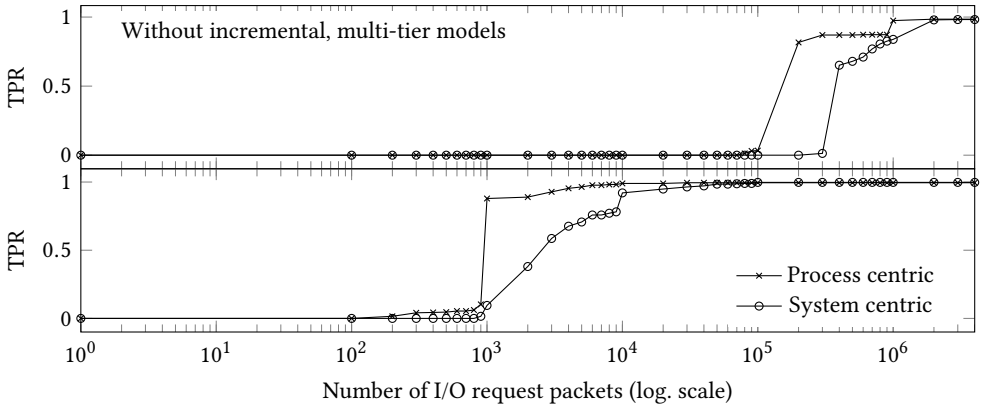


Fig. 5. 10-fold Cross Validation: TPR of process- and system-centric detectors, with and without the incremental, multi-tier approach. FPR ranges from 0.0 to 0.0015.

working ransomware samples and let them run for 60 minutes on the machines protected by SHIELDfS. This dataset (Table 6) is completely disjoint from the training dataset and was collected from VirusTotal as of May 2016. Interestingly, seven families (Locky, CryptoLocker, TorrentLocker, DirtyDecrypt, PayCrypt, Trolldesh, ZeroLocker) are not present in the training dataset.

Detection of Unseen Samples. SHIELDfS prevented malicious encryption in 100% of the cases, by restoring the 97,256 compromised files, and correctly detected 298 (97.70%) of the samples without any false positive. The top-tier, process-centric model contributed to detecting 95.2% of the samples, the incremental models were effective mainly in the case of ransomware performing code injections (4.3%), as expected. In one case, the incremental process-centric models identified the malicious process as suspicious and SHIELDfS invoked the system-centric model to take a final decision. CryptoFinder contributed to the detection of 69.3% of the samples.

Causes of False Negatives. Seven samples remained inactive for most of our analysis and encrypted just few files (less than 30). Fortunately, thanks to our conservative file-shadowing strategy, SHIELDfS had copied the original files, allowing their recovery. We investigated the cause of false negatives in the detection of cryptographic primitives and we found no evidence showing

Table 6. Dataset of 305 unseen samples of 11 different ransomware families.

Ransomware Family	No. Samples	Detection Rate
Locky	154 (50.5%)	150/154
TeslaCrypt	73 (23.9%)	72/73
CryptoLocker	20 (6.6%)	20/20
Critroni	17 (5.6%)	17/17
TorrentLocker	12 (3.9%)	12/12
CryptoWall	8 (2.6%)	8/8
Trolldesh	8 (2.6%)	7/8
CryptoDefense	6 (2.0%)	5/6
PayCrypt	3 (1.0%)	3/3
DirtyDecrypt	3 (1.0%)	3/3
ZeroLocker	1 (0.3%)	1/1
<i>Total</i>	305	298/305

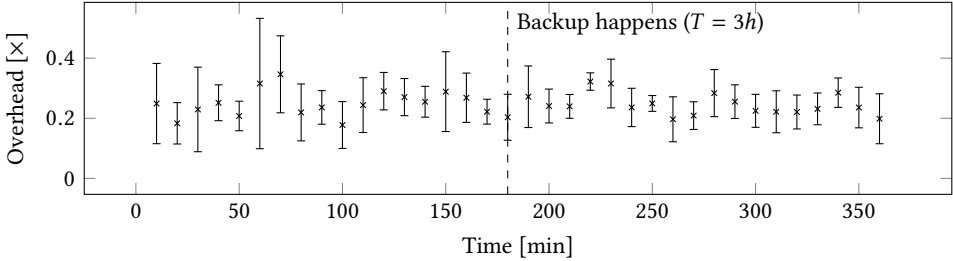


Fig. 6. Average (and standard deviation) perceived overhead introduced by SHIELDIFS on 5 real-users machines.

that the remaining samples were using AES. Therefore, we conclude that CryptoFinder’s detection capability of AES key schedule is 100%.

5.4 System Overhead

We evaluated the performance overhead and additional storage space requirements of SHIELDIFS.

User-Perceived Overhead. Our goal is to quantify, with good approximation, how much would SHIELDIFS slow down the typical user’s tasks, on average. To this end, we distributed to 5 users a new version of IRPLogger that collected file-size information in addition to the usual IRP logs.

Then, we reconstructed 6 hours worth of sequences of high-level system calls by analyzing about one month of low-level IRPs. For example, one `IRP_MJ_CREATE` followed by one or more `IRP_MJ_READ` corresponds to a `FileRead` call, and so on, by abstraction. Then, we estimated the perceived overhead for a user-level task as the average overhead due to all the filesystem calls executed during such task, taking into account the size of the affected files. We fixed 10 minutes as the duration of a user-level task, that is, while the user is interacting with the computer uninterruptedly. Figure 6 shows that the average estimated overhead is 0.26x. Indeed, we barely perceived it while using a machine protected by SHIELDIFS.

Runtime Overhead: Micro Benchmarks. We also evaluated the in-the-small performance impact of SHIELDIFS. We considered three sequences of filesystem operations on a series of 1,800 files of 18 varying sizes (from 1 KB to 128 MB): (1) open and read the files, (2) open and write them when they are not backed up already, and (3) open and write them when they are already backed up. We run each sequence 100 times on a Windows 10 machine equipped with a rotational hard disk drive, with and without SHIELDIFS, rebooting the machine after each test to avoid caching side effects. Figure 7 shows the overhead of each sequence. The overhead is higher (1.8–3.8x) when files need to be backed up, and remarkably lower (0.3–0.9x) when files are already backed up.

Table 7. Measured storage space requirements on real-users machines ($T = 3h$) and cost estimation considering \$3¢/GB.

User	Period [hrs]	Storage Required		Storage Overhead		Max Cost [USD]
		Max [GB]	Avg. [GB]	Max [%]	Avg [%]	
1	34	14.73	0.63	4.29	0.18	44.2¢
2	87	0.62	0.19	0.95	0.29	1.86¢
4	122	9.11	0.73	8.53	0.68	27.3¢
5	47	2.41	0.56	5.49	1.29	7.23¢
7	8	1.00	0.39	3.35	1.28	3.00¢

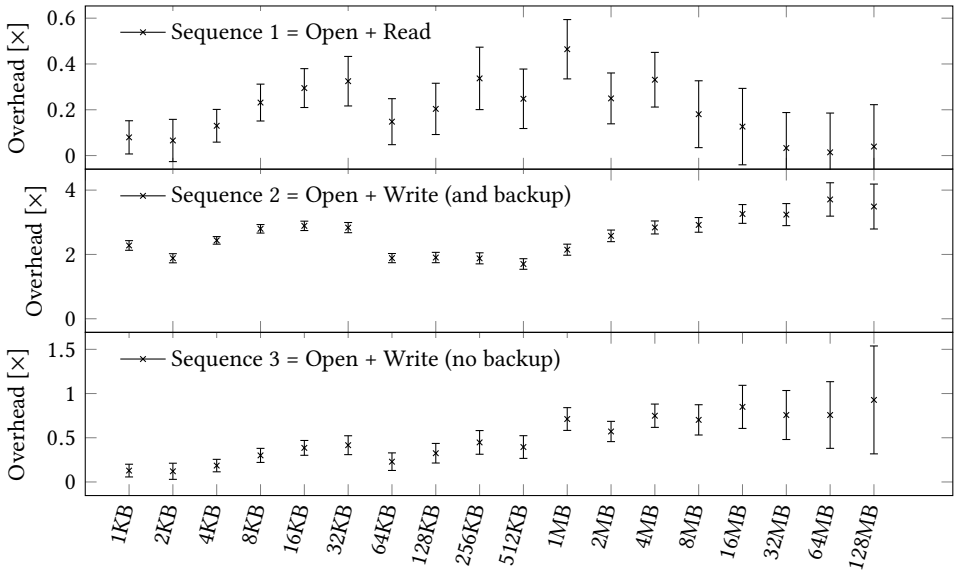


Fig. 7. Micro Benchmark: Average overhead.

Storage Space Requirements. During our experiments we kept track of the storage space required by SHIELDIFS to keep secure copies. Table 7 shows that with $T = 3h$, in the worst case (i.e., all files need to be backed up within T), SHIELDIFS requires 14.73 GB of additional storage space (i.e., \$44.2¢).

Parameter Setting. The T parameter determines how often SHIELDIFS creates copies of the files that require to be shadowed. Table 8 shows the average overhead and storage space required for $T \in [1, 4]$ hour(s) measured during our experiments. We can conclude that T does not significantly influence the overall performance overhead. Thus, as further discussed in Section 6, we advise to set it as high as to match the on-premise, long-term backup schedule.

6 DISCUSSION OF LIMITATIONS

From the results of our experiments we discuss the following list of limitations, in decreasing order of importance.

Susceptibility to Targeted Evasion. Ticks are essentially the “clock” of SHIELDIFS. At each tick, a decision is made. Since ticks are not based on time, but on the percentage of files accessed, an adversary may be interested in preventing to trigger the ticks, so to avoid detection. However, the only way to do it is to access zero or very few files, which is clearly against the attacker’s goal. Alternatively, in order not to cause a significant change in the feature values after code injection, an adversary may try to find an existing, *benign* host process that has *already* accessed about as

Table 8. Influence of T on runtime and storage overhead.

T [hrs]	Runtime Overhead		Storage Space Overhead			
	Avg [\times]	Std.dev [\times]	Max [GB]	Avg [GB]	Max [%]	Avg [%]
1	0.263	0.0404	5.4838	0.4040	4.353	0.586
2	0.262	0.0404	5.8402	0.4875	4.762	0.720
3	0.261	0.0403	5.5768	0.4994	4.522	0.746
4	0.260	0.0403	5.5927	0.5150	4.545	0.766

many files as the attacker wants to encrypt. This is very unlikely because, by design, such process can exist only if it has not already triggered the detection (otherwise SHIELDDFS would have already killed it already). That is, only if it has accessed a large number of files without violating the other features (e.g., mainly read operations, low entropy files). Assuming that the malware can find such a benign process to inject its malicious code, the process' features will start to change as soon as the malicious code will start encrypting the aforementioned files. At some point, the malicious code cannot avoid performing many write operations of high-entropy content.

If the malware knows precisely the thresholds of the classifiers and value of the parameter T , it could attempt to perform a mimicry attack [18] encrypting few files so as to remain below the thresholds until T hours. In this way, it will be identified as benign and the victim will lose the original copies. However to remain unaccountable, a ransomware cannot encrypt all the files in one round, so it would need to repeat this procedure every T hours. Setting T to large values will raise the bar, by forcing the attacker to wait for long. On the other hand, setting T very low guarantees that the recent (benign) modifications are accounted in the secondary drive. In this way, if a restore is needed, a very recent (up to T) copy is available. In other words, T allows to trade off mimicry resilience versus data freshness.

Multiprocess Malware. Ransomware injecting malicious code into many benign processes, each of them performing a small part of the malicious activity, could evade our detector if the attacker knows the feature values—which, is challenging for a userland malware. Multiprocess malware is partially mitigated by the combination of system-centric models with the incremental, multi-tier strategy. Nevertheless, ransomware could perform encryption very slowly. This however, is against the attackers' goal, who wants to encrypt all files before users can notice any change. Last, even if a malicious process is *not* detected, thanks to our conservative file-shadowing approach, a user noticing the encrypted files can manually restore the original files from the copies.

Cryptography Primitives Detection Evasion. A possible cause of false negatives of our approach is the use of dedicated ISA extensions of modern CPUs (e.g., Intel AES-NI [6]) to perform the encryption off memory, using a dedicated register file. However, in such case the malware binary code would contain those specific instructions, not to mention that the malware will work only if the victim machine supports the Intel AES-NI extensions.

The current proof-of-concept implementation of SHIELDDFS supports only the detection of AES. Supporting other ciphers is an implementation effort, as our approach is valid for the majority of symmetric block ciphers.

Tampering with the Kernel. SHIELDDFS runs in a privileged kernel mode. We implemented SHIELDDFS to be “non unloadable” at runtime, even by administrator users. Furthermore, SHIELDDFS is able to deny any operation that attempts to delete or modify the driver binaries. An administrator-privileged process, however, could try to prevent SHIELDDFS service from starting at boot, by modifying the Windows registry, and force a reboot. This limitation can be mitigated by embedding our approach directly in the kernel without the need for a service. Doing so, the only chance to bypass our system is to compromise the OS kernel.

Preventing Denial of Service. A malware could attempt to compromise SHIELDDFS itself by filling up the shadow drive. First, in this scenario it is likely that SHIELDDFS detects and stops the malicious process before it fills the entire space. Second, SHIELDDFS makes the shadow drive read-only, denying any modification request coming from userland processes. Last, SHIELDDFS could monitor the shadow drive and alert the user when it is running out of space.

7 RELATED WORKS

Kharraz et al. [9] studied the behavior of scareware and ransomware, observing its evolution during the last years, in terms of encryption mechanisms, filesystem interactions, and financial incentives. They suggested some potential defenses, but evaluating them was out of the scope of their paper. Indeed, while [9] analyzed the filesystem activity of ransomware, the authors (and any other work) did not focus on analyzing the filesystem activity of *benign* applications, which we found crucial to build a robust detector.

Concurrently and independently to our work, Kharraz et al. [8] and Scaife et al. [16] published two ransomware detection approaches, respectively UNVEIL and CryptoDrop. Although they both look at the filesystem layer to spot the typical ransomware activity, they do not provide any recovery capability. Also, their approaches do not include identification of cryptographic primitives. Differently from our work, UNVEIL includes text analysis techniques to detect ransomware threatening notes and screen lockers, along the line of [3], and CryptoDrop uses similarity-preserving hash functions to measure the dissimilarity between the original and the encrypted content of each file. These two techniques are complementary to ours, and can be added to SHIELDIFS as additional detection features.

Andronio et al. [3] studied the ransomware phenomenon on Android devices, proposing an approach, HelDroid, to identify malicious apps. Besides the difference in the target platform, HelDroid looks at how ransomware behaves at the application layer, whereas we focus on its low-level behavior. Thus, their approach is complementary to ours, also because it is based on static analysis.

Our data-collection and mining phase is somehow akin to what Lanzi et al. [11] did to perform a large-scale collection of system calls, with the purpose of studying malware behavior by means of the system and API call profiles. We focus on IRPs instead as they better capture ransomware behavior.

Lestringant et al. [12] applied graph isomorphism techniques to data-flow graphs in order to identify cryptographic primitives in binary code. Although [12] works at binary level, whereas SHIELDIFS identifies usage of cryptographic primitives at runtime, it is a valid alternative that can be used to complement our CryptoFinder.

8 CONCLUSIONS

In this paper, we proposed an approach to make modern operating systems more resilient to malicious encryption attacks, by detecting ransomware-like behaviors and reverting their effects safeguarding the integrity of users' data.

We foresee SHIELDIFS as a countermeasure that keeps an always-fresh, automatic backup of the files modified in the short term. We argue that, although older files can be asynchronously backed up with on-premise systems (because they have less strict time constraints), recent files may be of immense value for a user (e.g., time-sensitive content); even the loss of a small update to an important file may end up in the decision to pay the ransom, because the existing backup is simply too old. With traditional backup solutions alone there exist a trade off between performance, space and "freshness," not to mention that a ransomware may encrypt the backups as well! Generally, traditional solutions work well for incremental backups, long-term archives with no real-time constraints. Pushing such backup solutions to tighter time constraints while keeping reasonable system performance may result in side effects. For instance, once a file is encrypted by a ransomware, there exists a risk that it may replace the plaintext backup. Instead, SHIELDIFS works at a lower level. Thus, it is transparent to a ransomware that works at the filesystem's logical view. Therefore,

it is best suited for the protection of short-term file changes, leaving traditional backups protecting from long-term file changes.

REFERENCES

- [1] 2015. *Unlock the key to repel ransomware*. Technical Report. Kaspersky Lab.
- [2] 2016. Video demonstration of ShieldFS in action. (2016). <https://www.youtube.com/watch?v=0UlgdnQQaLM>
- [3] Nicoló Andronio, Stefano Zanero, and Federico Maggi. 2015. HelDroid: Dissecting and Detecting Mobile Ransomware. In *Research in Attacks, Intrusions, and Defenses*. Springer.
- [4] Liviu Arsene and Alexandra Gheorghe. 2016. *Ransomware. A Victim's Perspective*. Technical Report. Bitdefender. http://www.bitdefender.com/media/materials/white-papers/en/Bitdefender_Ransomware_A_Victim_Perspective.pdf
- [5] FBI. 2015. Criminals Continue to Defraud and Extort Funds from Victims Using CryptoWall Ransomware Schemes. (2015). <http://www.ic3.gov/media/2015/150623.aspx>
- [6] Shay Gueron. 2012. *Intel Advanced Encryption Standard (AES) New Instructions Set*. Technical Report. Intel. <https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf>
- [7] Microsoft Inc. 2014. File System Minifilter Drivers. (2014). [https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402(v=vs.85).aspx)
- [8] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*.
- [9] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*. Springer.
- [10] Vadim Kotov and Mantej Singh Rajpal. 2014. *Understanding Crypto-Ransomware: In-Depth Analysis of the Most Popular Malware Families*. Technical Report. Bromium.
- [11] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2010. AccessMiner: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM.
- [12] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 2015. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM.
- [13] Trend Micro. 2016. Ransomware Bill Seeks to Curb the Extortion Malware Epidemic. (2016). <http://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/ransomware-bill-curb-the-extortion-malware-epidemic>
- [14] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. 2012. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE.
- [15] Kevin Savage, Peter Coogan, and Hon Lau. 2015. *The evolution of ransomware*. Technical Report. Symantec.
- [16] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. 2016. CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.
- [17] Michele Spagnuolo, Federico Maggi, and Stefano Zanero. 2014. *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*. Chapter BitIodine: Extracting Intelligence from the Bitcoin Network.
- [18] David Wagner and Paolo Soto. 2002. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM.
- [19] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. 2015. Robust and Effective Malware Detection Through Quantitative Data Flow Graph Metrics. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.
- [20] Adam Young and Moti Yung. 1996. Cryptovirology: Extortion-based security threats and countermeasures. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE.