RWGuard: A Real-Time Detection System Against Cryptographic Ransomware: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings

Chapter *in* Lecture Notes in Computer Science · September 2018

DOI: 10.1007/978-3-030-00470-5_6

CITATIONS
90

READS
2,848

3 authors:

Shagufta Mehnaz
Purdue University West Lafayette
10 PUBLICATIONS **398** CITATIONS
SEE PROFILE

Anand Mudgerikar
Microsoft Security
17 PUBLICATIONS **518** CITATIONS
SEE PROFILE

Elisa Bertino
Purdue University West Lafayette
1,291 PUBLICATIONS **40,092** CITATIONS
SEE PROFILE

RWGuard: A Real-Time Detection System Against Cryptographic Ransomware

Shagufta Mehnaz(✉), Anand Mudgerikar, and Elisa Bertino

Purdue University, West Lafayette, IN, USA
{smehnaz,amudgeri,bertino}@purdue.edu

Abstract. Ransomware has recently (re)emerged as a popular malware that targets a wide range of victims - from individual users to corporate ones for monetary gain. Our key observation on the existing ransomware detection mechanisms is that they fail to provide an early warning in real-time which results in irreversible encryption of a significant number of files while the post-encryption techniques (e.g., key extraction, file restoration) suffer from several limitations. Also, the existing detection mechanisms result in high false positives being unable to determine the original intent of file changes, i.e., they fail to distinguish whether a significant change in a file is due to a ransomware encryption or due to a file operation by the user herself (e.g., benign encryption or compression). To address these challenges, in this paper, we introduce a ransomware detection mechanism, `RWGuard`, which is able to detect crypto-ransomware *in real-time* on a user's machine by (1) deploying decoy techniques, (2) carefully monitoring both the running processes and the file system for malicious activities, and (3) omitting benign file changes from being flagged through the learning of users' encryption behavior. We evaluate our system against samples from 14 most prevalent ransomware families to date. Our experiments show that `RWGuard` is effective in real-time detection of ransomware with zero false negative and negligible false positive (∼0.1%) rates while incurring an overhead of only ∼1.9%.

Keywords: Ransomware · Real-time detection · I/O monitoring

1 Introduction

Ransomware is a class of malware that has recently become very popular among cybercriminals. The goal of these cybercriminals is to obtain financial gain by holding the users' files hostage- either by encrypting the files or by locking the users' computers. In this paper, we focus on *crypto ransomware* which asks users for a ransom in exchange of decryption keys that can be used to recover the files encrypted by the attacker. Such a ransomware is now a significant threat to both individuals and organizations. Among the recent ransomware attacks, Petya [8] is the deadliest one; it affected several pharmaceutical companies, banks, at

M. Bailey et al. (Eds.): RAID 2018, LNCS 11050, pp. 114–136, 2018.
https://doi.org/10.1007/978-3-030-00470-5_6

least one airport and one U.S. hospital. Another massive ransomware that hit nearly 100 countries around the world is WannaCry [30]. This attack targeted not only large institutions but also any individual who could be reached. While ransomware has maintained prominence as one of the biggest threats since 2005, the first ransomware attack occurred in 1989 [12] and targeted the healthcare industry. The healthcare industry, which possesses very sensitive and critical information, still remains a top target.

Even though several techniques have been proposed for detecting malware, very few of them are specific to ransomware detection [6,10,13,14,16,26,27]. Such existing techniques, however, have at least one of the following limitations: (a) impractically late detection when several files have already been encrypted [13,26,27], (b) failure to distinguish benign file changes from ransomware encryption [6,10,13,14,16,26,27], (c) offline detection system that is unable to detect ransomware in real-time [13], (d) emphasis only on post-encryption phase which fails to recover files in most of the cases [16] or conflicts with secure deletion [6,10], and (e) monitoring applications' actions only for a limited amount of time after their installation [27].

Problem and Scope. In this work, we focus on the most critical requirement for a successful ransomware, i.e., *making the valuable resources (i.e., files, documents) unavailable to the user*, and design a solution, `RWGuard`, that protects against ransomware by detecting and stopping the ransomware processes at an early stage. Note that the ransomware families that lock the user's machine are out of the scope of this paper.

Approach. `RWGuard` employs three monitoring techniques: decoy monitoring, process monitoring, and file change monitoring. Unlike generic malware, ransomware wreak havoc systems within minutes (or seconds). Therefore, analyzing processes' file usage patterns and searching for ransomware-like behaviors result in delayed detections. To address this challenge, we strategically deploy a number of decoy files in the system. Since in the normal cases a decoy file should not be written, whenever a ransomware process writes to such a decoy file, our *decoy monitoring* technique identifies the ransomware process instantaneously. Though some research work [15,19] recommends using decoy files for detecting ransomware, such previous work does not present any analysis on the effectiveness of these decoy files with any real system design. *To the best of our knowledge, ours is the first work to empirically analyze the effectiveness of decoy techniques against ransomware.* The *process monitor* checks the running processes' I/O Request Packets (IRPs), e.g., IRP write, IRP create, IRP open, etc. While some existing approaches [13,14] are signature-based and look for specific I/O request patterns, we exploit the *rapid encryption property* of ransomware [10], use a number of IRP metrics for building baseline profile for each running process, and utilize these baseline profiles for performing process anomaly detection. The *file change monitor* checks all changes performed on the files (e.g., create, delete, and write operations) to determine anomalous file changes. From our experimental observations, we have found that monitoring only the process activities [13,14] or only the file changes [13,26] is not sufficient

for effective detection and results in both high false positives and high false negatives (e.g., we observed that the Cryptolocker ransomware encrypts files very slowly which sometimes evades process monitoring). *In this paper, we enhance these existing techniques and combine them with the decoy monitoring module in order to provide an effective solution for protection against ransomware.*

If a potential encryption of a file (not a decoy) is identified, the next step is to determine whether the file is encrypted by a ransomware (referred to as *ransomware encryption*) or by a legitimate user (referred to as *benign encryption*). Therefore, we also design a file classification mechanism that depending on the properties of a file, classifies the encryption as benign or malicious. In order to learn the user's file encryption behavior, we leverage an existing encryption utility (that utilizes cryptographic library CryptoAPI, e.g., Kryptel [17]) to be used by end-users and applications. Finally, our approach includes a mechanism that places hooks and intercepts calls to the functions in CryptoAPI library so as to monitor all benign file encryption.

Contributions. To summarize, `RWGuard` makes the following contributions:

1. A decoy based ransomware detection technique that is able to identify ransomware processes in real-time.
2. A ransomware surveillance system that employs both *process* and *file change* monitoring (to detect ransomware encrypting files other than decoy).
3. A classification mechanism to distinguish benign file changes from ransomware encryption by hooking relevant CryptoAPI functions and learning the user's file encryption behaviors.
4. An extensive evaluation of our ransomware detection system on 14 most prevalent ransomware families to date.

2 Background

Hybrid Cryptosystem. A hybrid cryptosystem allows the ransomware to use different symmetric keys for encryption of different files while using a single asymmetric key pair. The attacker generates the asymmetric public-private key pair on its own command and control infrastructure. The ransomware code generates a unique symmetric key for each file to be encrypted and then encrypts these symmetric keys with its public key. These encrypted symmetric keys are then left with the encrypted files. At this point, the user needs to pay the ransom to get the private key with which it can first retrieve the symmetric keys, and then decrypt the files.

IRPLogger. All the I/O requests by processes that are sent to device drivers are packaged in I/O request packets (IRPs). These requests are generated for any file system operation, e.g., open, close, write, read, etc. IRPLogger leverages a mini-filter driver [11] that intercepts the I/O requests. An example of IRPLogger entry is:

```
<Timestamp, PID, IRP/FastIO, Operation (READ/WRITE/OPEN/CLOSE/CREATE)>
```

CryptoAPI. CryptoAPI is a Microsoft Windows platform specific cryptographic application programming interface (API). This API, included with Windows operating systems, provides services to secure Windows-based applications using cryptography. It includes functionalities for encrypting (*CryptEncrypt*) and decrypting (*CryptDecrypt*) data, generating cryptographically secure pseudo-random numbers (*CryptGenRandom*), authentication using digital certificates, etc.

Microsoft Detours Library. Detours is a library for instrumenting arbitrary Win32 functions in Windows-compatible processors. It intercepts Win32 functions by re-writing the in-memory code for target functions. Detours preserves the un-instrumented target function (callable through a trampoline) as a subroutine for use by the instrumentation.

3 `RWGuard` Design

3.1 Threat Model

In our threat model, we consider an adversary that installs crypto-ransomware on victim machines through seemingly legitimate but malicious domains. We consider the operating system to be trusted. Ransomware generally targets and encrypts files that the user creates and cares about, and the user account already has all the privileges to access these files. However, though the assumption that ransomware executes only with user-level privileges seems reasonable (as otherwise, it may be able to defeat any existing in-host protection mechanisms, e.g., anti-malware solutions), this assumption does not apply to all the ransomware cases. We have observed some exceptions to this assumption where ransomware samples affect only a predefined list of system files and if not detected/terminated, gain root access, shut down the system, and at the next boot up, perform full disk encryption and ask for a ransom payment. Hence, we also include these ransomware samples in our threat model. Moreover, a malicious insider in an organization may gain the knowledge of decoy files and build a customized ransomware to sabotage the organization (installed as a logical bomb to detonate after the insider leaves the organization). A further discussion on how our `RWGuard` system handles such situations is given in Sect. 5.

3.2 Overview

Figure 1 shows the placement and the design overview of `RWGuard`. Any I/O request to the file system generated by any user space process first needs to be scheduled by the I/O scheduler. We leverage IRPLogger to fetch these system-wide file system access requests and parse those with our IRPParser.

`RWGuard` consists of five modules: (1) *Decoy Monitoring (DMon)* module, (2) *Process Monitoring (PMon)* module, (3) *File Change Monitoring (FCMon)* module, (4) *File Classification (FCls)* module, and (5) *CryptoAPI Function Hooking*

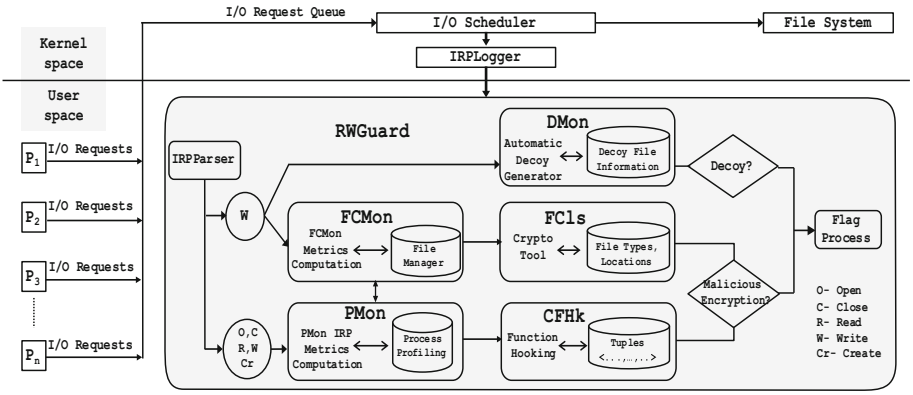

Fig. 1. Design overview of `RWGuard`

(CFHk) module. The *DMon* module considers only the IRP write requests as input and monitors whether there is any such request to a decoy file. The *PMon* and *FCMon* modules monitor process operations (IRP open, close, read, write, create) and file changes (IRP write), respectively. These two modules communicate in order to identify any process(es) making significant anomalous changes to the files. If such an event is identified, the *FCls* module checks the properties of the file and predicts the probability of the file change to be benign. Furthermore, the *CFHk* module checks whether a benign encryption (by the user) has been recorded for this file at the time of the file's significant change.

3.3 Decoy Monitoring (DMon) Module

The *DMon* module deploys decoy files that allow our system to identify a ransomware process in real-time. Since the decoy files should not be modified in normal situations, whenever a (ransomware) process tries to write such files, this module can immediately identify the process as malicious. Furthermore, the presence of a significant number of decoy files (though of smaller sizes) increases the probability that a ransomware would encrypt one of these files even before trying to encrypt an original file. Hence, the advantage of using decoy files is twofold: (1) it allows the detection system to readily identify a malicious process, and (2) it delays the time when ransomware starts encrypting the original files and thus gives enough time for anomaly detection to complete its analysis and stop the malicious processes even before they start encrypting the original files (see Sect. 5.2 for the experimental data about the time required by `RWGuard` to complete the analysis). `RWGuard` decoy files are generated with an automated decoy generator tool that we discuss in details in Sect. 4.2. Note that, our decoy generator periodically modifies the decoy files so that even if a ransomware looks at the time when a file is last modified (to ensure that the file it encrypts is valuable to the user), it would not be able to recognize the decoy files.

Table 1. Fast I/O read/write types

READ types
FASTIO_READ
FASTIO_MDL_READ
FASTIO_READ_COMPRESSED
FASTIO_READ_COMPLETE_COMPRESSED

WRITE types
FASTIO_WRITE
FASTIO_MDL_WRITE
FASTIO_MDL_WRITE_COMPLETE
FASTIO_WRITE_COMPRESSED
FASTIO_MDL_WRITE_COMPLETE_COMPRESSED

Table 2. Metrics for the *PMon* module

Metric #	Metric name
1	Number of IRP_WRITE requests
2	Number of FastIO_WRITE requests
3	Number of IRP_READ requests
4	Number of FastIO_READ requests
5	Number of IRP_OPEN requests
6	Number of FastIO_OPEN requests
7	Number of IRP_CREATE requests
8	Number of FastIO_CREATE requests
9	Number of IRP_CLOSE requests
10	Number of FastIO_CLOSE requests
11	Number of temporary file created

3.4 Process Monitoring (PMon) Module

Unlike some existing approaches [13,14] that look for *specific patterns* (e.g., read → encrypt → delete) in the processes' I/O requests, we exploit the fact that ransomware typically attempts to encrypt data rapidly [10] (to maximize damage and minimize the chance of being detected) which leads to anomalous numbers of IRPs. Exploiting this property results in faster detection since IRPs can be logged well ahead of actual file operations. Our *PMon* module monitors the I/O requests made by the processes running on the system. Though IRP is the default mechanism for requesting I/O operations, many ransomware perform file operations using fast I/O requests. Fast I/O is specifically designed for rapid synchronous I/O operations on cached files, bypassing the file system and the storage driver stack. Therefore, in our design, we monitor both the IRPs and the fast I/O requests. A fast I/O read/write operation can be any of the types listed in Table 1. Given that ransomware processes encrypt files rapidly, the behavior of such processes has certain characteristics. Hence, in this module, we train a machine learning model that given a process's I/O requests, identifies the process as benign or ransomware. Ransomware that encrypt files slowly may evade this module but are identified by the *FCMon* module as discussed in Sect. 3.5.

Process Profiling. In order to train the machine learning model, as a first step, we collect the IRPs (from this point, the term 'IRP' represents both I/O and fast I/O) of both benign and ransomware processes. Table 2 shows the IRP metrics used in this training phase which also includes the number of temporary files created by a process. The temporary files (.TMP) are usually created by ransomware to hold the data while copying or removing the original files. Once the profiles for benign and ransomware processes are built in the training phase, the *Process Profiling* component of the *PMon* module (Fig. 1) stores the model parameters to check against the running processes' parameters in real-time (i.e., the test phase). The *PMon* module re-computes the metrics listed in Table 2 for each running process over a 3 s sliding window.

Table 3. Performance evaluation for different machine learning techniques

Classifier	Accuracy (%)	ROC area	True positive rate	False positive rate	Precision	Recall
Naive Bayes	80.07	0.69	0.80	0.70	0.75	0.80
Logistic regression	81.22	0.72	0.81	0.66	0.77	0.81
Decision tree	89.27	0.87	0.89	0.18	0.89	0.89
Random forest	96.55	0.94	0.96	0.08	0.96	0.96

- **Training phase:** The data collection and classifier training steps are following:

1. **Data collection:** For the training set, we collect IRP data of processes from both ransomware samples and benign applications. We use nine of the most popular ransomware families, namely: Wannacry, Cerber, CryptoLocker, Petya, Mamba, TeslaCrypt, CryptoWall, Locky, and Jigsaw for the training phase. We also include benign processes, e.g., Explorer.exe, WmiPrvSE.exe, svchost.exe, FileSpy.exe, vmtoolsd.exe, csrss.exe, System, SearchFilterHost.exe, SearchProtocolHost.exe, SearchIndexer.exe, chrome.exe, GoogleUpdate.exe, services.exe, audiodg.exe, WinRAR.exe, taskhost.exe, drpbx.exe, lsass.exe, etc. It is important to note that most of the ransomware samples spawn multiple malicious processes during execution. Our final training dataset contains IRPs from 261 processes including both benign and malicious ones.
2. **Classifier training:** Using the training data, we train a machine learning classifier that, given a set of processes, is able to distinguish between ransomware and benign processes. In order to identify the best machine learning technique for this classification, we analyzed different classifiers, namely: *Naive Bayes* (using estimator classes), *Logistic Regression* (multinomial logistic regression model with a ridge estimator), *Decision Tree* [24], and *Random Forest* [3] classifiers. We used 10 fold cross validation on the obtained data set and measured accuracy, precision, recall, true positive rate and false positive rate for each of the above-mentioned classifiers. Table 3 presents a comparison of the classifiers used in our analysis. Figure 2 shows the results for all the classifiers in terms of ROC curves (which plot true positive rate against false positive rate). The low accuracy (∼80%) of the naive Bayes classifier can be attributed to its class independence property. From our observation, ransomware usually employs a combination of read, write, open, and close requests which are correlated. Therefore, assuming that these parameters are independent of each other leads to a lower accuracy. The regression classifier works slightly better than the naive Bayes classifier with an accuracy of ∼81%. A logistic regression model searches for a single linear decision boundary in the feature space. Hence, the low accuracy can be attributed to the

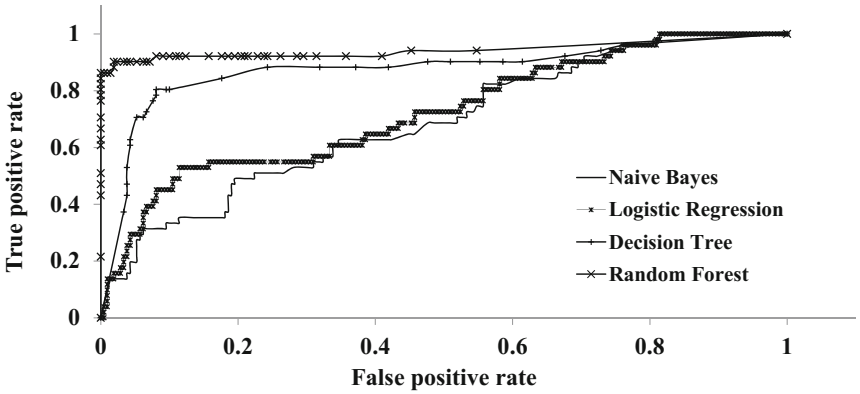

Fig. 2. ROC curves for different classifiers.

fact that our data does not have a linear boundary for decisions. The reason is that many ransomware make a large number of write/read requests as compared to the open/close requests. Therefore, the ideal decision boundary for our dataset would be non-linear.
The tree-based classifiers (random forest and decision tree) perform the best with accuracies of ∼97% and ∼89%, respectively. The reason is that the decision boundary for our data is non-linear and these classifiers build non-linear decision boundaries. However, the decision tree classifier is susceptible to over-fitting while random forest classifiers do not have this issue. Also, in terms of deployment, the random forest classifier is faster and more scalable compared to other classifiers. Therefore, finally, we use the random forest classifier in our `RWGuard` *PMon* module.

- **Test phase:** In the test phase, along with the nine families used for training, we add five more ransomware families in the experiment set: Vipasana, Satana, Radamant, Rex, and Matsnu. These samples are executed one at a time and depending on the spawned processes and their activities, the malicious processes are flagged. Details of the test phase results are given in Sect. 5.

File Encryption. In our experiments, we observe that few benign processes, e.g., Chrome, VMware tools are sometimes classified as malicious by the machine learning model due to these processes' I/O request behaviors. Therefore, besides monitoring the process profiling metrics, it is important to monitor whether a particular process is responsible for any significant file changes. Hence, our *PMon* module considers *file encryption* as a significant parameter (communicated by the *FCMon* module as described in Sect. 3.5) and identifies a process as malicious only if it encrypts files along with indications of anomalous I/O behaviors.

3.5 File Change Monitoring (FCMon) Module

This monitoring module can be configured to target a range of files from a single directory to the whole file system. It computes and stores the initial properties of the files (or, dynamically computes the properties when a file is created) and these properties are updated accordingly in the event of a file change. In real-time, the *FCMon* module looks for significant changes in those files after each write operation using the following metrics: (1) similarity, (2) entropy, (3) file type change, and (4) file size change. While some of these metrics have been used for ransomware detection in existing work [13,26], our goal is to verify the fast detections by the *PMon* module and thereby minimize the false positive rates. In what follows, we describe the *File Manager* component of the *FCMon* module and present the details of the above metrics.

File Manager. This component stores the current properties of each file (e.g., file type, current entropy of a file, file size, last modified time etc.) so that any significant change in the files' properties can be detected upon a write operation. If a new file is created, this component computes the properties of the new file instantly and stores them in the map (map key: file name and path, key value: computed properties).

Metrics. The metrics of *FCMon* module are following:

1. *Similarity* metric: In comparison with a benign file change, e.g., modifying some of the existing text or adding some text, an encryption would result in data that is very dissimilar to the original data. Therefore, the similarity between a file's previous (before the write operation) and later (after the write operation) versions is an important factor to understand the characteristics of the file change. In order to compute the similarity between two versions, we use *sdhash*, a similarity-preserving hash function proposed by Roussev et al. [25] for generating the file hashes. The *sdhash* function outputs a score in the range [0,100]. A score of 0 is obtained when we compute the similarity between two completely random arrays of data. Conversely, a score of 100 is obtained when we compute the similarity between two files that are exactly same. Hence, in the case of an encryption, this function outputs a value close to 0.
2. *Entropy* metric: Entropy, as it relates to digital information, is the measurement of randomness in a given set of values (data), i.e., when computed over a file, it provides information about the randomness of data in the file. Therefore, certainly, a user's data file in plaintext form has low entropy whereas its encrypted version would have a high entropy. Other than encrypted data, compressed data also has high entropy when compared to its plaintext form. A widely used entropy computation technique is Shannon entropy [21]. The Shannon entropy of an array of N bytes (assuming ASCII characters with values 0 to 255) can be computed as the following: $\sum_{i=0}^{255} P_i log_2 \frac{1}{P_i}$. Here, P_i

is the probability that a randomly chosen byte from the array is i, (i.e., $P_i = F_i/N$) where F_i is the frequency of byte value i in the array. This equation returns a value in the range of [0,8]. For an absolutely even distribution of byte values in the array, the output value is 8. Since encrypted files have bytes more evenly distributed (when compared to its plaintext version), the Shannon entropy significantly increases after encryption and results in a value near 8.

3. *File type change* metric: A file generally does not change its type over the course of its existence. However, it is common for a number of ransomware families to change the file type after encryption. Therefore, whenever a file is written, we compare the file types before and after the write operation.
4. *File size change* metric: Unlike file type change, file size change is a common event, e.g., adding a large text to a document. However, this metric along with other metrics can determine if the file changes are benign or malicious.

Upon detecting a file write operation that results in a file type change or exceeds at least one of the given thresholds for the metrics, that is, similarity (score < 50), or entropy (value > 6), and/or significantly changes the file size, the *FCMon* module shares the recorded metrics with the *PMon*, *FCls*, and *CFHk* modules for further assessment.

3.6 File Classification (FCls) Module

After the *PMon* and *FCMon* modules collaboratively identify a process responsible for anomalous I/O behavior and file changes, our detection system classifies whether the file is encrypted by the ransomware or the change is due to a benign operation. Our *FCls* module performs this classification by learning the usage of the crypto-tool (a utility leveraging CryptoAPI used for user's sensitive files' encryption and decryption, e.g., Kryptel [17]) and profiling the user's encryption behavior. For example, if a file is encrypted which is from the same directory and has the same type of a previously benignly encrypted file, this module assigns a higher probability for this file to be benignly encrypted (however, a ransomware cannot abuse this idea as described in *CFHk* module in Sect. 3.7). If the probability for a file is too low to belong to the benignly encrypted class and if the file gets encrypted, a flag is raised immediately by the *FCls* module. In order to remove false negatives (i.e., ransomware encrypts a file which has a high probability of being benignly encrypted), the encryption information is validated with the *CFHk* module which intercepts benign encryptions.

Protecting Sensitive Files: If at the time of the ransomware attack the sensitive files are already in encrypted form, the ransomware could further encrypt those files which makes those files unavailable too. Note that the *FCMon* module may not be able to flag this event with high probability. The reason is that the entropy would not change significantly since both the file versions (before and after the ransomware encryption) would have high entropy. To address such issue, we modify the permission settings for encrypted files, i.e., when a user encrypts a file using the crypto-tool, the only operations that we

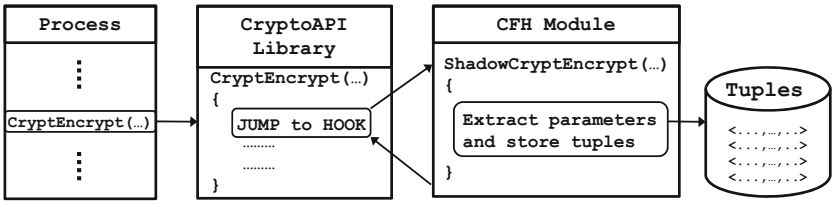

Fig. 3. CryptoAPI Function Hooking (CFHk) module

allow on that encrypted file are decryption and deletion (each of these operations requires the symmetric key used for encryption). Since it is impractical that someone would edit/modify an encrypted file before decryption, this permission setting suffices.

3.7 CryptoAPI Function Hooking (CFHk) Module

As described in Sect. 3.6, if the *FCls* module classifies the change to be the result of a possibly benign encryption, we need to further investigate whether the encryption was actually performed using the crypto-tool. Hence, the *CFHk* module places hooks at the beginning of the CryptoAPI library functions to redirect control of the execution to our custom-written functions. Figure 3 shows an example of hooking the 'CryptEncrypt' function included in the CryptoAPI library. Whenever a process calls the CryptEncrypt function to encrypt some file, the hook placed at the beginning of the CryptEncrypt function transfers control to a *shadow* CryptEncrypt function. This shadow CryptEncrypt function extracts a tuple $< key, algo, file, timestamp, process >$ for that particular call and stores this information in encrypted form for security purposes so that no other process can get access to this. The key for this encryption is derived from a secret password set by the user. Once the tuple is stored, the shadow CryptEncrypt function returns control to the original procedure, and the process continues its execution as if it had not been interrupted at all. The implementation details of this hooking procedure are discussed in Sect. 4.3.

To identify whether a file encryption is performed using the crypto-tool, we simply search 'CryptEncrypt' tuples that are captured by the *CFHk* module.

- If such a tuple is not found, we terminate the process that resulted in the file change so that no further encryption can take place.
- If such a tuple is found, the encryption is either benign (no action required) or a ransomware using CryptoAPI is responsible for the encryption. In the second case, we can recover all the files by using the *key* and *algo* information from the tuples (details in Sect. 5.4). Since in our system we also store the *file* information (by associating a *ReadFile* call with *CryptEncrypt*), we do not need to iterate over all the keys for a single file decryption which is an improvement over existing work [16].

Hence, the advantage of hooking the CryptoAPI library functions is twofold: (1) tracking all the benign encryption by the user, (2) recovering the ransomware encrypted files in the case that the ransomware dynamically links system-provided cryptographic libraries (i.e., Windows CryptoAPI).

4 `RWGuard` Implementation

4.1 IRPParser

While IRPLogger logs the I/O requests, the IRPParser component parses the log entries, extracts I/O requests, and provides these as input to the *DMon*, *PMon*, and *FCMon* modules accordingly.

4.2 Decoy File Generator

We have designed an automated decoy file generator tool that generates the decoy files based on the original file system and user preferences. By default, in each directory, it generates a decoy file with a name that is similar to one of the original files (selected at random or by the user depending on user preference) in that same directory so that the decoy files' names do not seem random to the ransomware. In order to make sure that the decoy files can be easily identified by the user, the naming options are selected based on the user's preferences which also makes the decoy files more unpredictable for the ransomware. The user is able to set different numbers of decoy files for different directories. In this way, the more sensitive files can be protected with a larger set of decoy files and also, manually setting the numbers makes it easier for the user to identify the decoy files during normal operations. The type extensions of the generated decoy files are: .txt, .doc, .pdf, .ppt, and .xls whereas the contents of the files are generated from the contents of neighboring files. Although we did not observe *selective* behavior (e.g., checking file name, file content, etc. before encryption) in any of the ransomware we experimented with, our decoy design is resilient to such future advanced ransomware. Note that the sizes of the decoy files in our system are randomly taken from a range (typically from 1 KB to few MBs) based on the sizes of the files in the original file system while the overall space overhead for decoy files is limited to 5% of the original file system size.

4.3 CryptoAPI Function Hooking

In our *CFHk* module, we leverage the Detours library introduced in Sect. 2. Detours hooks a function by moving a specific number of bytes (generally five bytes) from the beginning of the original function's memory address to the newly created hook function. In this blank space of the original function, an unconditional JMP instruction is added that would transfer the control to the hook function. The hook function then performs the necessary operations (e.g., safely storing the keys and other parameters passed to the original function). At the end

Table 4. Hooked CryptoAPI functions

Function	Details
CryptEncrypt	Encrypts data
CryptGenKey	Generates a random cryptographic session key or a public/private key pair
CryptDeriveKey	Generates cryptographic session keys derived from a base data value
CryptExportKey	Exports a cryptographic key/key pair from a CSP
CryptGenRandom	Fills a buffer with cryptographically random bytes

of these operations, another unconditional JMP instruction is added to transfer the control back to the original function. The compiled DLL file is placed into the registry key so that any process invoking the CryptoAPI functions would get hooked and our *CFHk* module would store information related to encryption. Table 4 lists the CryptoAPI functions we hook.

5 Evaluation

5.1 Experiment Dataset

While there exists different variants of ransomware, we build a comprehensive dataset from the most popular ransomware families: Locky, Cerber, Wannacry, Jigsaw, Cryptolocker, Mamba, Teslacrypt, Cryptowall, Petya, Vipasana, Satana, Radamant, Rex, and Matsnu. The ransomware samples are collected from VirusTotal [28], Open Malware [23], VXVault [29], Zelster [32], and Malc0de [22].

Note that among these samples, the first 9 families have been used in the training phase of the *PMon* module. However, we run each of these 14 ransomware samples (one at a time) in the detection phase to assess the detection effectivenesses and performance overheads of `RWGuard` modules. The reason behind not using the 5 samples for *PMon* module training is to measure how well this module performs with previously unseen ransomware samples.

5.2 Detection Effectiveness

We evaluate the performance of `RWGuard` by running the ransomware samples sequentially. Every time a ransomware sample is executed, we measure the time required for flagging each malicious process spawned by the ransomware. Once the ransomware is detected, we restore the system with a clean OS and execute the next ransomware sample.

Detection w/Decoy Deployment: We observe that ransomware detection with decoy deployment is extremely fast and ensures almost zero data loss. Note that the IRPParser component parses IRP logs collected in a 1 second cycle.

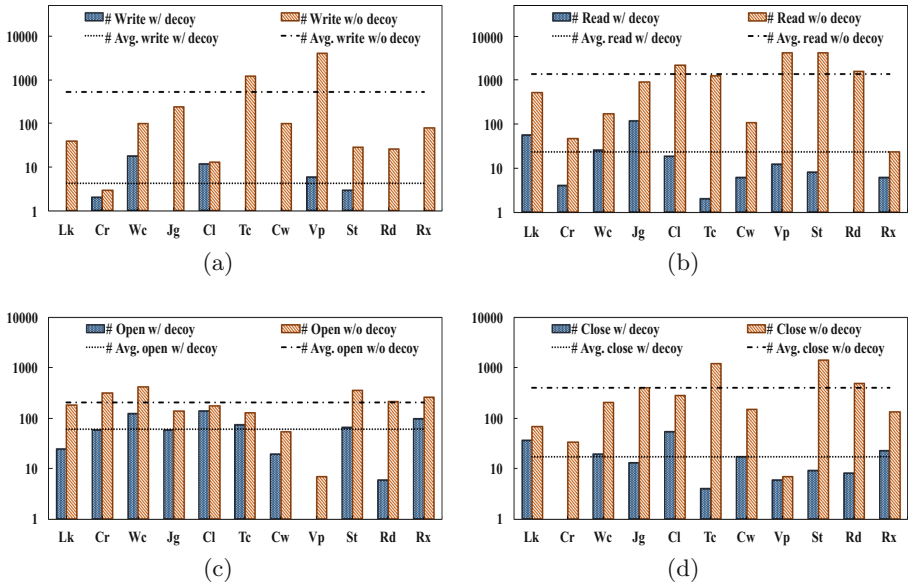

Fig. 4. Comparison between the cases of with and without decoy deployment in terms of the number of (a) *write*, (b) *read*, (c) *open*, and (d) *close* IRPs made by the ransomware samples until their detection (ransomware name abbreviations: Lk-Locky, Cr-Cerber, Wc-Wannacry, Jg-Jigsaw, Cl-Cryptolocker, Tc-Teslacrypt, Cw-Cryptowall, Vp-Vipasana, St-Satana, Rd-Radamant, Rx-Rex).

Therefore, with the decoy deployment, our system can identify a ransomware process right in the next cycle of the process's decoy file write request.

Figure 4 shows the comparison between the cases of with and without decoy deployment in terms of the number of write, read, open, and close IRPs (along with the average values for all the ransomware) made by the ransomware samples until their detection (in Figs. 4(a), (b), (c), and (d), respectively). The number of IRPs (for each IRP type) for each ransomware family is computed by running the samples at least 5 times. We find that with decoy deployment, for each of these IRP types, there is an improvement of at least one order of magnitude. Hence, the ransomware processes could be identified as soon as they start making IRP requests, i.e., in real-time. *For ransomware Locky, Jigsaw, Teslacrypt, Cryptowall, Radamant, and Rex, we observe that the first IRP write requests they make are for decoy files* (see Fig. 4*(a)) and thus are identified immediately. The Wannacry ransomware could make up to 18 IRP write requests (the highest) before it sends a write request for a decoy file (note that there can be multiple IRP write requests for a single file write operation).* An IRP write request is sent well ahead of the actual write operation and hence the actual number of files that can get encrypted before terminating the process is negligible (which also depends on the file size).

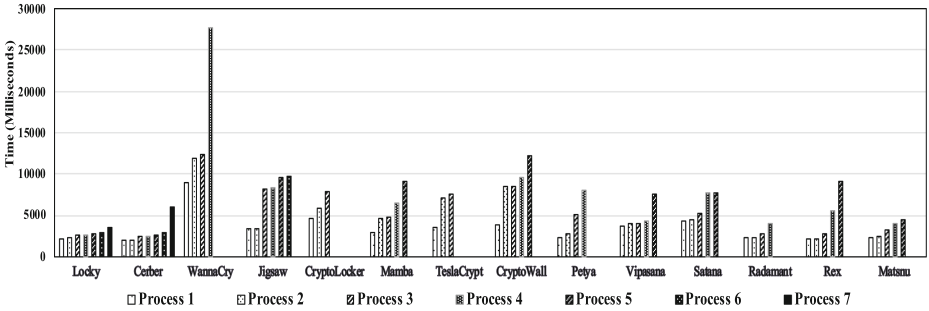

Fig. 5. Detection time required by `RWGuard` when there is no decoy deployment.

Note that, Fig. 4 does not show comparisons for the following three ransomware families: Mamba, Petya, and Matsnu. In our experiments, we have found that samples from these families affect only a predefined list of system files (and if there is no detection system activated except the decoy monitoring, this is followed by gaining root access, shutting down the system, and at the next boot up, performing full disk encryption and asking for a ransom payment). As a result, our *DMon* module cannot identify such ransomware families (however, the *PMon* and *FCMon* modules can) and therefore, we omit the comparison for these three families in this section.

Detection w/o Decoy Deployment: In order to further evaluate the effectiveness of `RWGuard`, we also consider an environment where there is no decoy file. This environment can be practical for the following two scenarios:

1. A ransomware encrypts only a predefined list of system files, i.e., even if the decoy files are deployed, the ransomware does not touch the decoy files (e.g., Mamba, Petya, and Matsnu ransomware families in our experiment dataset).
2. A malicious insider in an organization with the knowledge of decoy files' deployment can use customized ransomware to sabotage the organization and hold the ransomware responsible for this. Such an attack can be even launched as a logical bomb that can detonate after the insider has left the organization.

Figure 5 shows the time required to detect each of the samples (in milliseconds) while there is no decoy deployment in the system. The time computation starts when the ransomware sample is executed and ends when the corresponding process is flagged. Once the *PMon* and *FCMon* modules identify potential ransomware activity (i.e., malicious IRP/FastIO requests, significant file changes or encryption), the *FCls* and *CFHk* modules are communicated. If the file(s) that is (are) changed does (do) not belong to the 'benignly encrypted' class, and if there is (are) no corresponding encryption entry (entries) in the *CFHk* module, the process is immediately flagged. The average detection time for the first malicious processes spawned by all the ransomware is 3.45 s. However, we see that

all the ransomware spawn multiple malicious processes which are detected at different times by our monitoring system. We observe that the average required time for detecting all the spawned processes is 8.87 s. As we can see from Fig. 5, Locky, and Cerber spawn the highest number of malicious processes whereas CryptoLocker and TeslaCrypt spawn the lowest number of processes. According to our observation, most of the ransomware try to spawn processes with unique names or try to hide as system processes, e.g., explorer.exe. We also observe that different ransomware behave differently when the initially spawned malicious processes are killed by our system. For example, Wannacry sits idle for some time after the initial few processes are killed, before trying to spawn a new malicious process. This is the reason for the comparatively higher detection time for the last process in some of these ransomware.
Detection effectivenesses of different modules are discussed in the following:

- **Decoy Monitoring (DMon) module:** This module is the fastest to identify a ransomware process. Deploying a larger number of decoy files will result in even faster detection. For example, with a decoy generator that creates a shadow decoy file for each original file in the system, probabilistically, one out of each two write requests by a ransomware would belong to a decoy file.
- **Process Monitoring (PMon) and File Change Monitoring (FCMon) modules:** In most of the cases, the *PMon* module responded faster than the *FCMon* module in terms of flagging a malicious process. Even before the ransomware starts performing encryption, the *PMon* module is able to identify the malicious activities by monitoring the IRPs. In contrast, the *FCMon* module responds only after a file has been changed significantly. However, we observe that few benign processes, e.g., Chrome, VMware tools are sometimes misclassified as malicious by the *PMon* module due to these processes' I/O request behaviors. Therefore, it is important to also consider the analysis by the *FCMon* module to better understand whether a particular process is responsible for any malicious file changes and to remove any false positives.
- **File Classification (FCls) and CryptoAPI Function Hooking (CFHk) modules:** After the *PMon* and *FCMon* modules' detection that a process is making significant changes in the file(s), the information of the file(s) are sent to the *FCls* module which then computes the probability of these changes being benign. The false negatives of this module correspond to the cases in which the ransomware encrypts a file which has a high probability of being encrypted by the user benignly. Such false negatives are, however, detected by the *CFHk* module which identifies if the file is actually encrypted using the provided crypto-tool. With a 100% accuracy, the *CFHk* module can identify whether an encryption is performed by a ransomware or is a benign encryption. This module never flags a benign encryption. The only case of false positives (negligible, $\sim$0.1%) we have observed in the *FCls* and *CFHk* modules is when the user performs *file compression* in a directory for the first time. However, a first time benign file encryption in a directory is not flagged as malicious since the *CFHk* module can intercept the benign encryption operations. Note that the *FCls* and *CFHk* modules do not flag any process unless that process is identified as suspicious by one of the monitoring modules.

5.3 Size of Encrypted Data

In terms of the number of files, samples from ransomware families Locky, Jigsaw, Teslacrypt, Cryptowall, Radamant, and Rex could not encrypt any file with decoy deployment. The malicious processes for these families are identified on their first IRP write request. The numbers of IRP write requests made by ransomware families Cerber, Wannacry, Cryptolocker, Vipasana, and Satana before their detection are 2, 18, 12, 6, and 3, respectively, with decoy deployment. However, since an IRP write request is sent well ahead of the actual write operation and there can be multiple IRP write requests for a single file write, with decoy deployment, the average number of files lost is <1 with only Wannacry and Cryptolocker being able to encrypt 1 file each before their malicious processes are killed. The average number of IRP requests made by the ransomware families without any decoy deployment is $\sim$538 (with the strong assumption that the ransomware can evade the decoy deployment which is not the case for most of the families) whereas the average number of files affected is <10. Note that the number of files affected before detection depends not only on the number of IRP requests made but also on the time taken by a ransomware process to initiate the encryption routines (which is significant), type of encryption, size of the files, and the number of files the ransomware attempts to encrypt (this is because for each file the ransomware needs to generate a new key).

5.4 File Recovery

The *CFHk* module could recover all the files encrypted by the ransomware families: Locky, CryptoWall, and CryptoLocker. The encryption algorithms used by these samples are AES with CTR mode, AES in CBC mode, and AES, respectively. Note that the *CFHk* module in its current version cannot recover files that are encrypted using the ransomware's custom-written cryptographic library.

5.5 Performance Overhead

In the following, we discuss the performance overheads for different modules of `RWGuard`. The *DMon*, *FCls*, and *CFHk* modules have negligible overheads. The *DMon* module generates a single decoy file in each directory (if not set otherwise by the user) and randomly chooses the size of the decoy files from the range 1 KB–5 MBs while limiting the overall space overhead to 5% of the original file system size. At runtime, this module checks for decoy file write requests and modifies/regenerates the decoy files once per day at random times which has only a minimal overhead. The *FCls* module instantaneously classifies the files using file type and location information. The overhead for hooking a CryptoAPI function and computing and storing the corresponding tuple is a few milliseconds ($\leq$10 ms) which is negligible and thus cannot interrupt a user's normal operations.

Table 5. Memory overhead of `RWGuard`

Component	Memory consumed (KB)
Main Java module	14296
FCMon Entropy Calculator	7880
FCMon Similarity Index Calculator	5152
IRP Logger	42964

There is a main Java module which executes the IRPLogger, collects all the IRPs made in the system, parses the IRPs with IRPParser, and runs three parallel threads for *DMon*, *PMon*, and *FCMon* modules. The *FCMon* module consists of the components for computing the values of entropy and similarity index which use minimal CPU cycles since these are called only when there are write operations on the files. The memory usage of these components along with the main Java module is shown in Table 5. The average CPU usages for this main Java module and IRPLogger are 0.85% and 1.02%, respectively.

Overheads for Different Workloads. The performance overheads discussed above are recorded while running a web browser process and an integrated development environment (IDE) process along with regular operating system processes. However, in order to measure `RWGuard` detection performance and overheads for a heavy workload OS, we add several processes: two browsers (Chrome and Internet Explorer), two IDEs (Eclipse and PyChar), Windows Media Player, Skype, and other regular operating system processes. According to our experiments, this heavy workload does not significantly affect the time required by `RWGuard` for identifying ransomware processes while we have observed that IRPLogger and the Java module incur higher memory overhead (244456 KB and 45436 KB, respectively) due to this heavy workload. The detection time remaining unaffected by the heavy workload can be attributed to the fact that `RWGuard` fetches IRPLogger entries every 2 s which does not depend on the number of entries logged (the number of log entries is much higher for the heavy workload case). Since parsing the IRP logs is not an expensive operation, for the heavy workload case, the detection time is not significantly changed. Also, the memory overheads for the *FCMon* metrics' calculation remain similar.

5.6 Comparison with Existing Approaches

Table 6 presents a comparison among `RWGuard` and other exiting ransomware detection techniques with respect to monitoring, detection, and recovery strategies.

Table 6. Comparison of `RWGuard` with existing ransomware detection mechanisms

Solution	Real-time detection with decoy	Benign operation/ encryption profiling	File change monitoring	Process monitoring	Recovery of decryption key	Recovery of files
`RWGuard`	✓	✓	✓	✓	✓(partial)	✓(partial)
ShieldFS [6]	×	×	×	✓	×	✓
Unveil [13], CryptoDrop [26], Redemption [14]	×	×	✓	✓	×	×
PayBreak [16]	×	×	×	×	✓(partial)	✓(partial)
EldeRan [27]	×	×	×	✓	×	×
FlashGuard [10]	×	×	×	×	×	✓

6 Discussion and Limitations

Novelty. To the best of our knowledge, `RWGuard` with the decoy technique is the first system with very fast real-time (few milliseconds) detection capabilities. Even without the decoy deployment, the other monitoring modules are able to minimize the damage by identifying the ransomware processes at the time of their I/O requests. An average of 538 I/O write requests within the average detection time of 3.45 s shows how rapidly a ransomware attempts to encrypt the user's files while `RWGuard` exploits this property to terminate the ransomware at an early stage. Also, whereas the existing approaches are unable to distinguish benign file changes from malicious ones, the *FCls* module along with the *CFHk* module is able to overcome such false positives.

Inevitability. Our robust decoy design makes it impossible for the ransomware to recognize a decoy file by any of its properties. The ransomware would need to install some spyware and monitor the file activities in the system in order to determine which ones are modified by the end-users and applications and which are executed by our decoy tool. Moreover, obfuscation techniques can be used to make difficult for the ransomware to analyze the applications in order to determine which application is the decoy generator. Our integrated monitoring modules, *PMon* and *FCMon*, employ scrutiny on metrics that are inclusive of any malicious activity by the ransomware. For example, a smart ransomware that encrypts files slowly would still be detected by the *FCMon* module. While the monitoring modules *DMon*, *PMon*, and *FCMon* do not let a ransomware activity remain undetected (i.e., they prevent false negatives), the *FCls* and *CFHk* modules distinguish benign file operations from malicious ones (i.e., they prevent false positives). Hence, we argue that independently of the intelligence of modern ransomware, `RWGuard` raises the evasion bar for ransomware significantly.

File Recovery. Note that, the *CFHk* module monitors all (benign and ransomware) file encryption that leverage 'CryptoAPI' functions. Therefore, if a ransomware leveraging CryptoAPI library (3 of the 14 ransomware families that we have analyzed use this library) becomes successful in encrypting a set of

files before our early detection, using the hooking mechanism, we can retrieve the parameters (including the decryption keys) of those specific cryptographic function calls and consequently restore the encrypted files. Our experiments (Sect. 5.4) show that the *CFHk* module is able to recover the files encrypted by the 3 ransomware families with a 100% success rate. The rest of the ransomware samples experimented in our evaluation did not use CryptoAPI but their custom-written cryptographic library. Moreover, code obfuscation is a common technique used by the modern ransomware families. Obfuscation strategies, such as incremental packing and unpacking, make it more difficult to identify cryptographic primitives in the ransomware binary. While there are techniques (e.g., [31]) that look for cryptographic operations in the process memory, we have not incorporated those in our system due to their huge performance overhead.

Limitations. While the *DMon* module is quick in identifying a malicious process, the *PMon* and *FCMon* modules are anomaly based and hence probabilistically bound to miss some of the malicious activity. Also, these modules are based on the logging of IRP calls and file activity. The time lag between logging these activities and parsing them for anomalies provide a small window for the ransomware to perform its malicious activities as discussed in Sect. 5.3.

7 Related Work

Detection Techniques. Kharraz et al. [13–15] propose systems that monitor the I/O request patterns of applications for signs of ransomware-like behaviors. Scaife et al. [26] have designed *CryptoDrop*, a system that alerts users during suspicious file activity, e.g., tampering with a large amount of the user's data. Sgandurra et al. [27] propose *EldeRan*, a machine learning approach that monitors actions performed by applications in their first phases of installation and checks for characteristics signs of ransomware. Lee et al. [18] propose a ransomware prevention mechanism based on abnormal behavior analysis in a cloud system. Cabaj et al. [4] present a software-defined networking (SDN) based detection approach that utilizes the characteristics of ransomware communication. Andronio et al. [1] propose a technique to detect Android ransomware that applies to only mobile platforms- where applications are analyzed in-depth before they are released in any app market. Huang et al. [9] propose a measurement framework for end-to-end of ransomware payments. In contrast, `RWGuard` is the fastest solution that identifies ransomware infection in real-time with decoy techniques, prevents malicious processes from making changes to the files, and also determines the original intent of file changes.

Post-encryption Techniques. Kolodenker et al. [16] propose a system, called *PayBreak*, that intercepts system provided crypto functions, collects and stores the keys, and thus, can decrypt files only for the ransomware families that use system provided crypto functions. Continella et al. [6] propose the *ShieldFS* tool that monitors low-level file system activity to model the system over time. Whenever a process violates these models, the affected files are transparently rolled

back. However, it requires shadowing a file whenever it is modified and thus incurs high overhead. FlashGuard, a system developed by Huang et al. [10] leverages the fact that SSD performs out-of-place writes and thus holds the invalid pages for up to 20 days to perform data recovery after ransomware encryption. However, this type of recovery methods conflict with the idea of secure deletion and may result in privacy issues and data leakage. Given the limitations of the existing post-encryption recovery techniques, it is of uttermost importance that faster detection techniques be developed against ransomware.

Decoy Techniques. Decoy techniques have been previously proposed to defend against insider threats [2]. Though some research work [15,19] recommends using decoy files for detecting ransomware, such previous work does not include any analysis on the effectiveness of the decoy files. *Randomly generated decoy* files in commercial solutions (e.g., [7]) are susceptible of detection by sophisticated ransomware. Moreover, unlike `RWGuard`, their decoy files are deployed during the installation process which simply leaves the files unmodified for a long time and thus makes these files less interesting for the ransomware. Also, it is not clear how these solutions would handle special ransomware families, e.g., Mamba, Petya, and Matsnu, that affect only a predefined list of system files.

Cryptographic Primitives Identification Techniques. Discovering cryptographic primitives in a given binary is another research direction where crypto-ransomware including cryptographic operations could be identified beforehand [5,31]. Calvet et al. [5] developed such a technique and evaluated the performance of their system on a set of known malware samples. Lestringant et al. [20]'s approach to obtaining the similar goal leverages graph isomorphism techniques. Although these approaches could identify cryptographic primitives in obfuscated programs, their poor performance makes them impractical for real-time defense even with the most recent work [31] resulting in a 5-6X slowdown in average.

8 Conclusions and Future Work

In this paper, we introduce `RWGuard` that detects crypto-ransomware on a user's machine in real-time while removing the false positives due to the user's benign file operations. We evaluate `RWGuard` against 14 most prevalent ransomware families. Our experiments show that `RWGuard` is effective in early detection of ransomware with only negligible false positives ($\sim$0.1%) and zero false negatives while incurring an overhead of only $\sim$1.9%. Furthermore, `RWGuard` recovers all files that are encrypted using CryptoAPI by the corresponding ransomware. As part of the future work, we plan to profile other existing encryption libraries and in real-time scan the process's memory for similar operations so that we can recover the keys used for encryption and restore the files. Moreover, we plan to take snapshots of the ransomware processes' memories before terminating the processes and analyze those for traces of encryption/decryption keys.

Acknowledgement. We thank our shepherd, Alina Oprea, and the anonymous reviewers for their valuable suggestions. The work reported in this paper has been supported by the Schlumberger Foundation under the Faculty For The Future (FFTF) Fellowship.

References

1. Andronio, N., Zanero, S., Maggi, F.: HelDroid: dissecting and detecting mobile ransomware. In: Bos, H., Monrose, F., Blanc, G. (eds.) RAID 2015. LNCS, vol. 9404, pp. 382–404. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26362-5_18
2. Bowen, B.M., Hershkop, S., Keromytis, A.D., Stolfo, S.J.: Baiting inside attackers using decoy documents. In: Chen, Y., Dimitriou, T.D., Zhou, J. (eds.) SecureComm 2009. LNICST, vol. 19, pp. 51–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05284-2_4
3. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
4. Cabaj, K., Gregorczyk, M., Mazurczyk, W.: Software-defined networking-based crypto ransomware detection using HTTP traffic characteristics. CoRR abs/1611.08294 (2016)
5. Calvet, J., Fernandez, J.M., Marion, J.Y.: Aligot: cryptographic function identification in obfuscated binary programs. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 169–182. ACM, New York (2012). https://doi.org/10.1145/2382196.2382217
6. Continella, A., et al.: ShieldFS: a self-healing, ransomware-aware filesystem. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, pp. 336–347. ACM, New York (2016). https://doi.org/10.1145/2991079.2991110
7. CryptoStopper: www.watchpointdata.com/cryptostopper/
8. Fox-Brewster, T.: Petya or notpetya: why the latest ransomware is deadlier than wannacry. FORBES, June 2017. https://www.forbes.com/sites/thomasbrewster/2017/06/27/petya-notpetya-ransomware-is-more-powerful-than-wannacry
9. Huang, D.Y., et al.: Tracking ransomware end-to-end. In: Proceedings of the 2018 IEEE Conference on Security and Privacy, SP 2018 (2018)
10. Huang, J., Xu, J., Xing, X., Liu, P., Qureshi, M.K.: Flashguard: leveraging intrinsic flash properties to defend against encryption ransomware. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, pp. 2231–2244. ACM, New York (2017)
11. Microsoft Inc.: File system minifilter drivers, May 2014. https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402(v=vs.85).aspx
12. Jayanthi, A.: First known ransomware attack in 1989 also targeted healthcare. Beckers Hospital Review, May 2016. http://www.beckershospitalreview.com/healthcare-information-technology/first-known-ransomware-attack-in-1989-also-targeted-healthcare.html
13. Kharaz, A., Arshad, S., Mulliner, C., Robertson, W., Kirda, E.: Unveil: a large-scale, automated approach to detecting ransomware. In: 25th USENIX Security Symposium (USENIX Security 2016), pp. 757–772. USENIX Association, Austin (2016)

14. Kharraz, A., Kirda, E.: Redemption: real-time protection against ransomware at end-hosts. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) Research in Attacks, Intrusions, and Defenses. LNCS, pp. 98–119. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66332-6_5
15. Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., Kirda, E.: Cutting the gordian knot: a look under the hood of ransomware attacks. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) DIMVA 2015. LNCS, vol. 9148, pp. 3–24. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20550-2_1
16. Kolodenker, E., Koch, W., Stringhini, G., Egele, M.: Paybreak: defense against cryptographic ransomware. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS, pp. 599–611. ACM, New York (2017). https://doi.org/10.1145/3052973.3053035
17. Kryptel: https://www.kryptel.com/products/kryptel.php
18. Lee, J.K., Moon, S.Y., Park, J.H.: CloudRPS: a cloud analysis based enhanced ransomware prevention system. J. Supercomput. **73**(7), 3065–3084 (2017). https://doi.org/10.1145/3052973.3053035
19. Lee, J., Lee, J., Hong, J.: How to make efficient decoy files for ransomware detection? In: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, pp. 208–212. ACM, New York (2017)
20. Lestringant, P., Guihéry, F., Fouque, P.A.: 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, ASIA CCS, pp. 203–214. ACM, New York (2015). https://doi.org/10.1145/2714576.2714639
21. Lin, J.: Divergence measures based on the shannon entropy. IEEE Trans. Inf. Theor. **37**(1), 145–151 (2006). https://doi.org/10.1109/18.61115
22. Malc0de: http://malc0de.com/rss
23. Malware, O.: http://openmalware.org
24. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc. (1993)
25. Roussev, V.: Data fingerprinting with similarity digests. In: Chow, K.-P., Shenoi, S. (eds.) DigitalForensics 2010. IAICT, vol. 337, pp. 207–226. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15506-2_15
26. Scaife, N., Carter, H., Traynor, P., Butler, K.R.B.: Cryptolock (and drop it): stopping ransomware attacks on user data. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), pp. 303–312, June 2016. https://doi.org/10.1109/ICDCS.2016.46
27. Sgandurra, D., Muñoz-González, L., Mohsen, R., Lupu, E.C.: Automated dynamic analysis of ransomware: benefits. Limitations and use for detection, ArXiv e-prints, September 2016
28. VirusTotal: https://www.virustotal.com
29. VXVault: http://vxvault.siri-urz.net/URL_List.php
30. Wong, J.C., Solon, O.: Massive ransomware cyber-attack hits nearly 100 countries around the world. Theguardian, May. https://www.theguardian.com/technology/2017/may/12/global-cyber-attack-ransomware-nsa-uk-nhs
31. Xu, D., Ming, J., Wu, D.: Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In: Proceedings 2017 IEEE Symposium on Security and Privacy, pp. 129–140, May 2017
32. Zelster: https://zeltser.com/malware-sample-sources/