Limits of I/O Based Ransomware Detection: An Imitation Based Attack

Chijin Zhou*, Lihua Guo*, Yiwei Hou*, Zhenya Ma*, Quan Zhang*,
Mingzhe Wang*, Zhe Liu†, and Yu Jiang*✉
*BNRist, School of Software, Tsinghua University, Beijing, China
†Computer Science and Technology, NUAA, Nanjing, China

***Abstract*—By encrypting the data of infected hosts, cryptographic ransomware has caused billions of dollars in financial losses to a wide range of victims. Many detection techniques have been proposed to counter ransomware threats over the past decade. Their common approach is to monitor I/O behaviors from user space and apply custom heuristics to discriminate ransomware. These techniques implicitly assume that ransomware behaves very differently from benign programs in terms of heuristics. However, when we investigated the behavior of benign and ransomware programs, we found that the boundary between their behaviors was blurred. A ransomware program can still achieve its goal even though it follows the behavior patterns of benign programs. In this paper, we aim to explore the limits of ransomware detection techniques that based on I/O behaviors. To this end, we present ANIMAGUS, an imitation-based ransomware attack that imitates behaviors of benign programs to disguise its encryption tasks. It first learns behavior patterns from a benign program, and then spawns and orchestrates child processes to perform encryption tasks behaving the same as the benign program. We evaluate its effectiveness against six state-of-the-art detection techniques, and the results show that it can successfully evade these defenses. We investigate in detail why they are ineffective and how ANIMAGUS is different from existing ransomware samples. In the end, we discuss potential countermeasures and the benefits that detection tools can gain from our work.**

1. Introduction

Cryptographic ransomware is designed to encrypt the data of infected hosts and demand payment from victims for decryption. It has been widespread and caused the loss of billions of dollars [1] and even human lives [2]. Over the past decade, research on ransomware has gained significant traction in academia and industry. As ransomware spreads, researchers collect a large number of ransomware samples to conduct analysis on its behaviors [3], [4], evolution [5], [6], and payment transactions [7].

Similar to traditional malware defense, the best way to fight against ransomware is to detect and terminate it. Therefore, many detection techniques [8], [9], [10], [11], [12], [13], [14] have been proposed in the past decade. Their typical approach is monitoring I/O behaviors from user space and applying their custom heuristics to discriminate malicious behaviors. They usually consider several key features to detect if a process is malicious. For example, some of them [9], [10], [11], [13] consider the number of read/write/delete operations performed and the entropy of buffers written by the process as part of their key features. The rationale is that a ransomware program rapidly performs a number of operations to encrypt files or erase original contents, and often writes high-entropy buffers to files due to the nature of encryption algorithms.

These detection techniques implicitly assume that ransomware behaves very differently from benign programs regarding their custom features. Based on that, one can adopt a rule-based or machine learning-based approach to detect malicious behaviors. For example, Redemption [10] manually assigns weights to its custom features and calculates the overall malice score for every process at runtime; ShieldFS [9] trains a random forest model based on its custom features and then applies the model at runtime to discriminate malicious behaviors. Similarly, many other detection techniques leverage their proposed features and report perfect (i.e., 100%) detection accuracies and negligible false positive rates on existing ransomware samples [5].

The assumption, however, does not necessarily hold because ransomware and benign programs can behave very similarly. Take the feature of read/write/delete operation as an example. If the number of these operations performed by a program exceeds a given threshold, it will be marked as malicious. However, benign programs like the compressor or compiler can also perform a large number of these operations within a short interval. On the other hand, a ransomware program can split its short-duration encryption tasks and run these subtasks intermittently. This makes it more difficult to discriminate ransomware programs although increases the total time of encryption tasks. Therefore, although existing heuristics seem to delineate behavior boundaries between the benign and the malicious, ransomware programs can still cross the boundary and perform attacks. Existing detection techniques [9], [10], [11] realize this issue and leverage some combined features to mitigate it, but this idea has limited effect on making ransomware attacks infeasible.

We investigated the I/O behaviors of benign and ransomware programs and observed that their patterns may

✉. Yu Jiang is the corresponding author.

overlap. We first analyzed existing ransomware samples and broke down their I/O behaviors to files into three patterns: (1) *Read-Write*: a file is accessed by a read operation and then accessed by one or more write operations without any delete operation; (2) *Read-Delete*: a file is accessed by a read operation and then deleted from disks; and (3) *Only-Write*: a file is accessed by one or more write operations without any read or delete operation. We regard these three patterns as building blocks of ransomware attacks because they can be combined to encrypt a file. For example, a ransomware program can overwrite a file with encrypted data using Read-Write pattern; or the program can delete a file and write an encrypted version to another file using a combination of Read-Delete and Only-Write patterns. We then investigated widely-used benign programs and analyzed their dynamic I/O behaviors. An interesting observation is that many of their accesses to files also conformed to these three patterns. For example, during a ten-minute execution, FireFox accessed 1,095 files in Read-Write pattern, 63 files in Read-Delete pattern, and 1,495 files in Only-Write pattern. This observation indicates that ransomware and benign programs may behave similarly. In other words, *a ransomware program can still achieve its encryption goal even though it follows the same I/O behavior patterns as benign programs*. Since detection techniques rely heavily on I/O behaviors, an attack that imitates benign programs may fail these techniques.

In this paper, we propose an imitation-based ransomware attack to help existing detection techniques realize the limits of their feature engineering. The limits do not imply the ineffectiveness against existing ransomware. Instead, we argue that these limitations stem from the fact that the customized features used by these techniques may not be necessary for ransomware attacks. Even though a combination use of features is able to discriminate all existing ransomware, an attacker can still craft an exception as long as the features are not necessary. As a proof of concept, our proposed attack is designed to imitate behaviors of benign programs in order to disguise its encryption tasks.

The attack consists of two modules: *benign behavior analysis* and *attack orchestration*. Before any attack begins, the *benign behavior analysis* module abstracts a behavior template from real-world I/O behaviors. It first runs a benign program to collect fine-grained behaviors. Next, it analyzes the I/O behaviors of each file to check if they match one of the three patterns. As a result, it generates a behavior template for future attacks. During attacks, the *attack orchestration* module acts in the same behavior pattern as the behavior template and encrypts files. It first queries file metadata from the victim's filesystem. Next, it orchestrates encryption tasks by considering both the metadata and the template and emits a list of instructions for each subprocess. Finally, it spawns subprocesses according to the template, and the subprocesses follow their own instruction list to access files and write encrypted data.

We implement a prototype of the proposed attack, referred to as ANIMAGUS, and evaluate its effectiveness against six state-of-the-art detection techniques, including Kaspersky Total Security [15], 360 Total Security [16], Windows Defender [17], ShieldFS [9], Unveil [8] and Redemption [10]. Results show that although these techniques can identify malicious behaviors of existing ransomware samples, our attack can still elude them. We investigate in detail why the detection techniques are ineffective and how ANIMAGUS is different from existing ransomware samples. In the end, we discuss the potential countermeasures and the benefit that existing techniques can gain from our work.

This paper makes the following contributions:

- We identify limits of existing ransomware detection techniques. Their I/O behavior-based detection classifiers may be ineffective, because a ransomware program can behave similarly to benign programs while still achieving its encryption goal.
- We design and implement ANIMAGUS, an imitation-based ransomware attack, as a proof of concept. It first learns a behavior template from a benign program and then orchestrates subprocesses to execute encryption tasks during its attack. The relevant artifacts will be available at https://github.com/ChijinZ/Animagus.
- We evaluate the proposed attack. Our experimental results highlight that the attack can imitate various benign programs and successfully elude existing defense techniques.

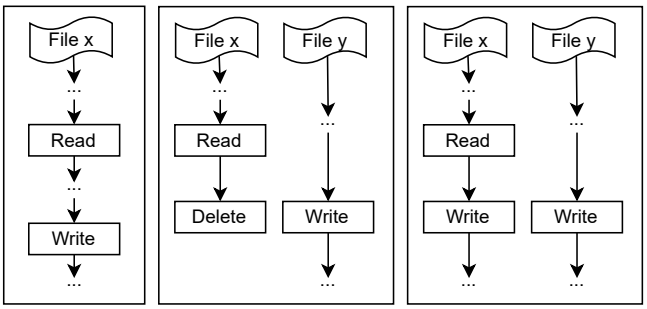
2. Background and Related Work

2.1. Cryptographic Ransomware Attack

Cryptographic Ransomware is a malware category that encrypts infected hosts' files and demands a ransom payment for the decryption of the files. This type of extortion is imposed by exploiting the victim's fear of losing valuable data or locking up critical resources. It has impacted not only servers and personal computers but also all computational systems, including smartphones, IoT/CPS devices, and many others [18]. The victims span not only ordinary end-users, but also governments and business organizations in almost all sectors [5]. Due to its notoriety, ransomware has gained significant traction in academic research as well as in industry. Many research efforts are paid to analyze its behaviors [3], [4], [19], evolution [5], [6], payment transactions [7], and social impacts [20].

Powered by modern *hybrid cryptosystem* [21], attackers can ensure that victims cannot decrypt locked files without the decryption key. In a ransomware attack, an attacker first generates an asymmetric public-private key pair on their own command and control (C&C) server. Next, on the victim's machine, the ransomware code generates a unique symmetric key for each file, called *session key*, and encrypts each file using the session key. In the end, every session key is encrypted with the attacker's public key and left together with the encrypted file contents. After receiving ransom payments, the attacker first decrypts the session key using the private key, and then decrypts files using the corresponding session key.

The lifecycle of ransomware attacks can be generalized into four phases: (1) *Infection*: the attacker delivers



(a) overwrite (b) delete and rewrite (c) smash and rewrite

Figure 1: Three types of ransomware's encryption tasks. (a) ransomware overwrites the target file with encrypted data. (b) ransomware first reads the file, and then deletes it and writes encrypted data to another file. (c) ransomware first reads the target file, and then smashes it with random data filled and writes encrypted data to another file.

ransomware to a victim system; (2) *Communication*: the ransomware connects to its C&C server to obtain necessary information (e.g., encryption keys); (3) *Execution*: the ransomware performs encryption tasks to lock files of the victim system; (4) *Extortion*: the ransomware notifies the victim and urges payment by displaying a ransom note. Once attacked, victims can only follow the ransom note to get their files back. A ransom note usually includes instructions on how to purchase cryptocurrencies, e.g., Bitcoin, and ransom addresses that victims are expected to pay into [7]. In terms of infection and communication, ransomware is similar to traditional malware, which has been well studied by previous work [22], [23], [24], [25], [26], [27].

In this paper, we focus on the execution phase because it is when actual malicious actions take place. During the phase, a ransomware attack should read file contents, perform cryptographic operations, write encrypted information, and most importantly, erase the original content. Therefore, a ransomware attack has many interactions with the victim's filesystem. Figure 1 presents three types of ransomware's encryption tasks summarized by previous studies [8], [14]. An encryption task is defined as a sequence of operations for encrypting a file. Note that these operations can be performed by multiple processes. To erase the original content of a file, ransomware either deletes the file directly (Figure 1b), or messes up the content (Figure 1a and 1c). In the former case, file contents might not be wiped out from the disk; thus, victims have a chance to recover their files without paying the ransom. However, file contents are securely erased in the latter case, which makes recovery almost impossible without the assistance of the attacker.

2.2. Detection of Cryptographic Ransomware

Ransomware detection techniques can be categorized as static and dynamic analysis. Traditional static malware detection techniques such as binary feature engineering [28] and signature matching [29], [30] have also been applied to ransomware detection. Nevertheless, some studies [31] have shown that obfuscation tricks can evade the static analysis. Therefore, studies on ransomware detection [8], [9], [10], [11], [12], [13], [14] pay more attention to dynamic analysis. Among all detection techniques, the I/O behavior is the most widely used characteristic since ransomware performs malicious actions on the victim's file system [5]. So a common approach is monitoring I/O behaviors from user space and applying their custom strategies to discriminate malicious behaviors.

A detection tool typically considers several key features to help detect malicious I/O behaviors. There are six features widely used in detection tools: (1) *Write Entropy*: data encrypted by ransomware usually follows a random distribution due to the nature of encryption algorithms, thus resulting in high entropy values; (2) *File Type Coverage*: ransomware generally accesses a large number of files of specific types within a short time interval; (3) *Directory Traversal*: ransomware greedily traverses the filesystem looking for target files; (4) *Read Files*: ransomware must read from lots of files; (5) *Write Files*: ransomware must write encrypted data to a large number of files; (6) *Delete Files*: ransomware may delete original files to erase data. Each detection tool also has its own unique proposed features to further improve detection accuracy. With these features, a tool can adopt a rule-based or machine learning-based approach to check if a process behaves maliciously. For example, Redemption [10] manually assigns weights to features and calculates overall malice score for every process at runtime; ShieldFS [9] collects a large number of I/O behaviors from benign programs and ransomware to train a random forest model as a classifier, and then applies the classifier at runtime to check malicious behaviors. Many detection strategies reported a perfect (i.e., 100%) detection accuracy with a negligible false positive rate [5].

3. Attack Overview

3.1. Threat Model

Our attack uses the same threat model described in previous research on ransomware [10], [11], [14]. We only focus on the execution phase of ransomware attacks in this paper. So we assume that our malicious code is running on victim systems. This can happen in several scenarios, e.g., our program is directly started by the victim, delivered by a drive-by download attack, or installed via a malicious email attachment. How to get victim systems infected is out of our research scope. We also assume that our malicious code has privileges to access files like any other user-level programs such as text editors, media players and web browsers. We make no further assumptions about victim systems, which means that the operating systems are trusted and have any ransomware detection tools pre-installed. In addition, we assume that encryption algorithms, e.g., AES, are reliable, so that encrypted data cannot be brute-forced in a reasonable amount of time. Overall, this will be a realistic threat model. It minimizes the requirements for a ransomware attack: the malicious code only has access to files. An attacker does

not need elevated privileges to run as administrator and run in kernel mode like a malicious kernel driver would.

Attack goal: The goal of our attack is to encrypt desired files, e.g., images and documents, on victim systems. We consider an attack successful when it encrypts all desired files on the victim system without being detected by existing ransomware detection tools. In addition to encryption, our program should also be able to decrypt encrypted files after the attack.

3.2. Motivation

The encryption tasks of ransomware have been well studied and summarized by previous work [8], [14]. As Figure 1 shows, the encrypted data can be either overwritten into the original file or rewritten into a new file. Meanwhile, the original file is erased by direct deletion or by overwriting it with other data. We conclude that the ransomware will access the file in one of the following three exploitable patterns:

- *Read-Write*: a file is accessed by a read operation and then accessed by one or more write operations without any delete operation;
- *Read-Delete*: a file is accessed by a read operation and then deleted from disks;
- *Only-Write*: a file is accessed by one or more write operations without any read or delete operation.

We regard these three patterns as building blocks of ransomware attacks because they can be combined to encrypt a file. For example, a ransomware program can overwrite a file with encrypted data (as Figure 1a) using Read-Write pattern; the program can also delete a file and write an encrypted version to another file (as Figure 1b) using a combination of Read-Delete and Only-Write patterns; or the program can smash file contents and write an encrypted version to another file (as Figure 1c) using a combination of Read-Write and Only-Write patterns. For simplicity, we omit other I/O operations such as opening, querying directory information, and querying file metadata, because they are less critical in ransomware attacks and detection.

We first investigated I/O behaviors of ten widely-used benign programs, including web browsers (Microsoft Edge, FireFox and Chrome), document editors (WPS Office and Microsoft Office), file compressors (7Zip and WinRAR),

TABLE 1: The number of files accessed in each pattern during ten minutes of regular use of benign programs.

Program	# files accessed in Read-Write pattern	# files accessed in Only-Write pattern	# files accessed in Read-Delete pattern
MS Edge	252	517	61
FireFox	1,095	1,495	63
Chrome	254	459	39
WPS Office	33	106	44
MS Office	28	78	181
7Zip	0	197	3,831
WinRAR	9	482	777
Golang-go	191	450	164
Rustc	78	1,569	733
Visual Studio	93	162	837

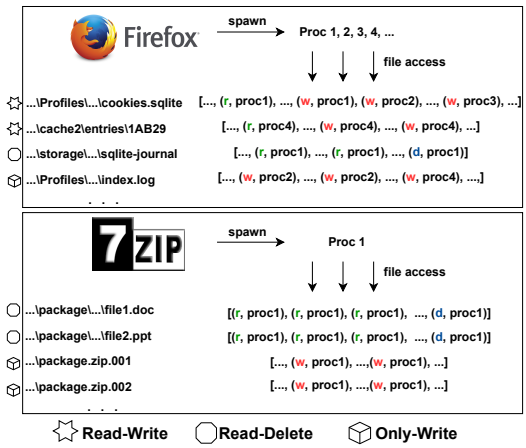


Figure 2: Demonstration of how the three exploitable patterns exist in I/O behaviors of FireFox and 7Zip.

compilers (Golang-go and Rustc), and integrated development environments (Visual Studio). We collected behaviors by running each program for ten minutes with real-world workloads, e.g., using browsers to randomly visit websites, using document editors to randomly open and edit some *.doc* files, and using file compressors to pack and unpack archives. TABLE 1 presents the statistics of these programs' I/O behaviors. Although they behaved quite differently, they all accessed a number of files in three exploitable patterns. Figure 2 demonstrates I/O behaviors of FireFox and 7Zip. When users visit websites, FireFox will access a large number of files, such as cookies, caches and logs, etc. A total of 3,281 unique files were accessed in ten minutes of normal use. Among them, 1,095 files, 1,495 files and 63 files can be regarded as accessed in Read-Write, Read-Delete, and Only-Write patterns, respectively. For example, accesses to *cookie.sqlite* and cache files follow Read-Write pattern; accesses to log files follow Only-Write pattern. 7Zip, on the other hand, does not access files in Read-Write pattern. Most of the time, it only reads file contents and stores compressed data to its output archive files. It also deletes files if users enable the `delete-after-compression` flag.

We then collected 232 ransomware samples from public repositories [32], [33], [34] and investigated their I/O behaviors. We observe that ransomware behave quite differently from benign programs, and it can be identified using some naive features like access frequency and entropy of written buffers. Ransomware samples tend to greedily encrypt files within a short period, and write many high-entropy buffers to files. For example, a sample of WannaCry, an infamous ransomware family, wrote buffers to 22,037 files in 152.45 seconds and over 70% of written buffers were considered high-entropy. Everything they perform, while designed for the efficiency of encryption, makes them different from benign programs and is very easy to detect.

In summary, we observe that benign programs execute a number of operations that can be used as building blocks for ransomware attacks. This indicates that *a ransomware*

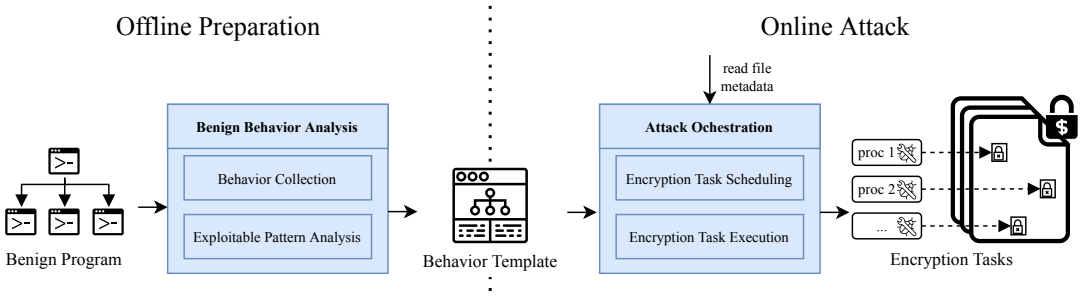


Figure 3: Overall design of our attack. During the offline preparation phase, the *benign behavior analysis* module first collects dynamic I/O behaviors of a benign program and then analyzes the access pattern of each file. As a result, it abstracts a behavior template from the behaviors. During the online attack phase, the *attack orchestration* module reads file metadata from the infected victim system, receives the behavior template, and then schedules encryption tasks based on them. Finally, our attack program spawns subprocesses to execute encryption tasks, behaving the same as the benign program.

program can still achieve its encryption goal even though it follows I/O behavior patterns of benign programs. Existing ransomware programs have not taken this into consideration, making them easy to spot. Since existing detection techniques rely heavily on I/O behaviors, an attack that imitates benign programs' behaviors has the opportunity to elude these techniques.

3.3. Imitation-Based Attack

Motivated by the above observation, we propose an *imitation-based ransomware attack*, which imitates behaviors of benign programs to disguise its encryption tasks. It is designed to help existing techniques fine-tune their heuristics. We define the imitation here as a transformation from a sequence of collected benign behaviors $B = \langle b_i(f_i)\rangle_{i\in[1,N]}$ to the one $B' = \langle b'_i(f'_i)\rangle_{i\in[1,N]}$ that implicitly contains encryption tasks on victim files V. Each behavior $b_i(f_i)$ is an I/O operation to a file f_i. The benign behaviors are collected during an offline preparation phase; the transformed ones are used to guide an online ransomware attack on the victim system. We define a file f in benign behaviors as an *exploitable file* if the access sequence $\langle b_j(f)\rangle_{b_j\in B}$ to the file conforms to the three exploitable patterns. Exploitable files need to be combined to establish encryption tasks. For example, Read-Delete and Only-Write exploitable files should be combined to an encryption task, so that the attack program can read contents from a victim file and delete it, and then write encrypted contents into another file. Therefore, some of exploitable files will not be used to establish encryption tasks if no combinable files are left. We define T as a set of exploitable files used in encryption tasks. Our imitation strategy is to follow the same I/O sequence as benign behaviors while replacing the accesses to files in T with encryption operations to victim files. Formally, the transformation rule is

$$trans(b_i(f_i)) = \begin{cases} b_i(d_i) & \text{if } f_i \notin T \\ b'_i(v_i) & \text{if } f_i \in T \end{cases}$$

where $v_i \in V$ is one of the files on the victim system to be encrypted; $d_i \in D$ is a dummy file that is aimlessly created and accessed by our program on the victim system; b'_i is an I/O operation, slightly different from b_i, but ensures the access $\langle b'_j(v)\rangle_{b'_j\in B'}$ to encrypt the file or erase original contents for every $v \in V$. Take Read-Write exploitable pattern as an example. If the access sequence to an exploitable file reads all contents by process p_1 at time t_1, writes 1,024 bytes with entropy etp_2 by process p_2 at time t_2, and writes 4,096 bytes with entropy etp_3 by process p_3 at time t_3, then our program will access a victim file in a similar way and at similar intervals, but will write encrypted data to the file. In this way, we can ensure that our attack program achieves encryption goals while performing I/O accesses similar to those of a benign program.

4. Attack Design

In this section, we detail the design of the proposed attack. As shown in Figure 3, it consists of two main modules: a *benign behavior analysis* module to collect imitable behaviors, and an *attack orchestration* module to perform encryption tasks. The first module runs on the attacker's C&C server before any attack begins, and the second module runs on victim systems to achieve the ransomware's goal.

We consider the whole process in two phases: the offline preparation phase and the online attack phase. During the offline preparation phase, the *benign behavior analysis* module first collects real-world I/O behaviors by running benign programs. To obtain fine-grained I/O information, it collects behaviors from a custom kernel driver. Next, it analyzes the access behaviors of each file to check if the access matches one of the exploitable patterns. Finally, it outputs a behavior template for future attacks. During the online attack phase, the *attack orchestration* module first requests the behavior template from the C&C server and reads file metadata on the victim system. Next, it schedules encryption tasks according to the template and spawns subprocesses to execute them. Finally, the subprocesses access files according to the task. Below we detail the design of these two modules and ultimately demonstrate how victims' files are encrypted.

Time	Proc	Operation	File	Extra Info
T_0 + 0	P1	QUERY INFO	F1	Null
T_0 + 12	P1	QUERY INFO	F2	Null
T_0 + 14	P2	QUERY INFO	F2	Null
T_0 + 20	P2	OPEN	F2	mode: ALL_ACCESS
T_0 + 25	P2	READ	F2	buff len: 4096
T_0 + 30	P2	WRITE	F2	buff len: 1024 entropy: 5.67
. . .				

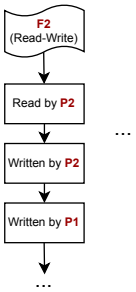


Figure 4: An example of behavior template. It mainly includes a series of records with placeholders and a list of file placeholders accessed in one of the exploitable patterns. Placeholders (highlighted in red) will be instantiated with concrete values during online attacks.

4.1. Benign Behavior Analysis

This module runs on the attacker's C&C server before any attack begins. The goal is to learn a behavior template from a benign program to facilitate heavily-disguised online attacks. As Figure 4 shows, a behavior template mainly includes two parts: (1) a series of records with placeholders, each indicating how and when a process accesses a file; and (2) a list of file placeholders accessed in one of the exploitable patterns. In addition, it also includes minor information such as the hierarchy of processes. After receiving a behavior template, a ransomware program can instantiate the template by assigning concrete values to placeholders based on the victim system's environment. Two sub-components are responsible for the behavior template generation: (1) *Behavior Collection* for collecting real-world behaviors of benign programs and abstracting behavior templates from them; and (2) *Exploitable Pattern Analysis* for analyzing access behaviors of each file to check if they match one of the exploitable patterns.

Behavior Collection. There are many approaches for collecting behaviors of running programs, for example, Process Monitor [35] on Windows and strace [36] on Linux. However, they are not flexible enough for our scenario because fine-grained information such as entropy of written buffers can not be obtained. Therefore, we develop the behavior collection part as a kernel driver and intercept the message sent to the disk device, which offers more comprehensive I/O information than existing tools. On Windows, all I/O requests sent to device drivers by user-space processes are packaged in the format of I/O request packets (IRPs). We thus intercept these IRPs from user-space processes and log necessary information in our kernel driver. Other information like hierarchies of processes are also recorded to help the imitation during attacks. After collecting IRPs, the behavior template will thus be abstracted from the IRPs. The processes in the IRPs are grouped by their process IDs and replaced with placeholders. In the same way, files are grouped and replaced by their file paths. Finally, a series of records with placeholders are generated, each indicating how and when a process accesses a file.

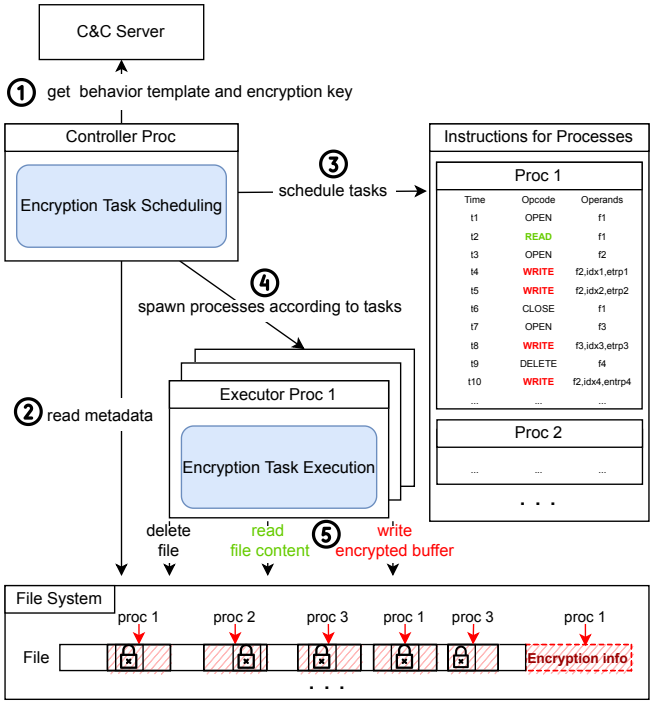


Figure 5: Demonstration of online attack. At first, the main controller process obtains necessary information (step ① and ②), and schedules encryption tasks based on the received behavior template (step ③). Next, It spawns subprocesses in chronological order and assigns an instruction list to each (step ④). Finally, the subprocesses access files following their own instruction list and thus encrypt files (step ⑤).

Exploitable Pattern Analysis. As explained in Section 3.2, a file used by ransomware is necessarily accessed in one of the three exploitable patterns, i.e. *Read-Write*, *Read-Delete* and *Only-Write*. Combining these exploitable patterns can establish an encryption task to a file. However, if collected behaviors do not include enough files accessed in the exploitable patterns, our malicious code cannot imitate those behaviors and perform encryption tasks. Therefore, we analyze the access behaviors of each file in the collected IRPs to check if the access pattern matches one of the exploitable patterns. Note that the three exploitable patterns are mutually exclusive, and thus access behaviors of a file can only be categorized into one of the patterns. Besides, the access behaviors of a file may come from different processes. For example, *F2* is accessed by *P1* and *P2* in Figure 4. After the pattern matching, we check if the number of exploitable files exceeds a threshold. If there are few exploitable files, we consider that this benign program cannot be used as an imitation target for our attacks.

4.2. Attack Orchestration

This module is designed to conduct real cryptographic ransomware attacks on victim systems. Figure 5 demonstrates the procedure of our online attack. First, similar to

other malware, our program communicates with its C&C server. It obtains a behavior template and an encryption key from the server. Second, it queries file metadata, i.e., the path and size of every victim file, from the victim's filesystem. Third, it schedules encryption tasks by considering both the metadata and the behavior template, and emits a list of instructions for each subprocess. Fourth, it spawns subprocesses following the same hierarchy of process as the behavior template. Finally, the subprocesses follow their own instruction list to access files and write encrypted data.

Our program randomly selects a certain percentage (e.g., 50%) of blocks in a file to encrypt. Victims still cannot decrypt locked files without a decryption key because (1) the blocks are selected randomly, so the victims have no way of knowing which parts are encrypted; and (2) it is impossible to decrypt these blocks in a reasonable amount of time because of the nature of modern ciphers. This encryption strategy brings two main advantages. First, multiple processes can synergistically encrypt the same file. Once the blocks to encrypt have been determined, each process can use a block cipher to independently encrypt the blocks it is responsible for. Without this strategy, our program cannot imitate a benign program with multiple processes. Second, each process can write encrypted contents with low entropy. Due to the nature of encryption algorithms, encrypted data usually follow a random distribution and are therefore high-entropy. However, if the buffer written each time is a combination of encrypted and original contents, then this buffer will not be highly randomized. Based on this encryption strategy, our program can schedule multiple processes to execute the encryption task. We will detail the design of the scheduling and execution in the following paragraphs.

Encryption Task Scheduling. Our program begins making a plan for encryption after obtaining behavior templates and file metadata. Algorithm 1 presents this procedure. The first thing is to combine exploitable files in the behavior template to establish encryption tasks. The accesses to exploitable files are categorized into three exploitable patterns, i.e., Read-Write, Read-Delete and Only-Write. We can combine them to establish two types of encryption tasks: (1) *Delete-Rewrite*: our program can delete a file and write an encrypted version to another file using the combination of Read-Delete and Only-Write patterns; and (2) *Overwrite*: our program can overwrite a file with encrypted data using Read-Write pattern. Line 3-11 presents how our program extracts encryption tasks from the behavior template. The number of extracted encryption tasks is the number of files our program is able to encrypt in one cycle. Our program will run multiple cycles until all victim files are encrypted.

After establishing encryption tasks, our program assigns a task to each victim file, and forms instruction lists for subprocesses. This procedure is an instantiation of the behavior template. In one cycle, given an encryption task, a file is selected to match it (Line 18). The selection considers two factors: the file size and the number of write operations in the encryption task. Next, our program instantiates related records of the behavior template for this encryption task

Algorithm 1: Encryption Task Scheduling

```
   Input  : Behavior template B
1  encryptTasks ← getEmptySet()
2  expFiles ← getExploitableFiles(B)
   // extract Delete-Rewrite tasks
3  for rdExpFile in getReadDeleteFiles(expFiles) do
4      if not hasOnlyWriteFiles(expFiles) then
5          break
6      owExpFile ← popOnlyWriteFile(expFiles)
7      encryptTasks ∪ = {(rdExpFile, owExpFile)}
8  end
   // extract Overwrite tasks
9  for rwExpFile in getReadWriteFiles(expFiles) do
10     encryptTasks ∪ = {(rwExpFile)}
11 end
12 while true do
       // a mapping of a process to its instructions
13     M ← getEmptyMap()
       // read a certain number of metadata from FS
14     F ← readFileMetadata(len(encryptTasks))
15     if isEmpty(F) then
16         break
       // assign an encrypt task to each file
17     for task in encryptTasks do
18         file ← popSuitableFile(F, task)
19         records ← getRecords(B, task)
20         instrs ← instantiate(records, file)
21         speedupReplay(instrs)
22         updateMap(M, Instrs)
23     end
24     waitForTaskExecution(M)
25 end
```

(Line 19-20). It replaces the file placeholders with the actual file path. For example, if a file whose path is `/home/file` matches an Overwrite task, then our program will obtain all records related to the task's Read-Write file placeholder, and replace the placeholder with `/home/file`. Besides, by modifying the execution time of each instruction, our program can control subprocesses to replay the imitated behaviors at a self-adjusting speed (Line 21). It adjusts the speed according to the number of encryption tasks in order to maintain a reasonable throughput during attacks. The strategy of self-adjusting speed is detailed in Appendix A.3. After the path replacement and the replay speedup, our program generates a set of instructions for this encryption task. The instruction aims to instruct a specific subprocess when and how to access a specific file. Next, it spawns subprocesses and waits for execution of encryption tasks to complete (Line 24).

Encryption Task Execution. Based on instructions scheduled by the controller process, subprocesses are spawned and ready to access files. Note that processes exchange necessary information through IPC channels, e.g., encryption key or file contents. Algorithm 2 presents how each subprocess executes their instructions. First, a subprocess requests an encryption key and cipher (e.g., AES256) from the controller process (Line 1). Next, it executes its instructions one by one. Every instruction has a timestamp indicating when this instruction is supposed to be executed. Therefore, before executing an instruction, it must wait until the time is up (Line 3). If the instruction is a read operation,

Algorithm 2: Task Execution of Each Process

```
   Input  : Instruction list of this process L
1  cipher, key ← requestEncryptionInfo()
2  for Inst in L do
3      waitUntil(Inst.time)
4      if Instr.Opcode == Read then
5          content ← readFile(Inst.file)
6          broadcastContent(Inst.file, content)
7      else if Instr.Opcode == Write then
8          content ← requestContent(Inst.file)
9          buf ← content[Inst.startIdx..Inst.endIdx]
           // select blocks to keep buffer low-entropy
10         blocks ← selectBlocks(buf)
11         encptBlocks ← encrypt(cipher, key, blocks)
12         encptBuf ← injectBlocks(buf, encptBlocks)
13         writeFile(Inst.file, Inst.startIdx, encptBuf)
14     else
15         execute(Inst)
16 end
```

the subprocess will read the file and broadcast the file contents (Line 4-6). If the instruction is a write operation, the subprocess will write encrypted data to the file. It will request file contents from other processes (Line 8), and get a sliced buffer from the content according to the instruction (Line 9). The subprocess will encrypt the sliced buffer and write it back to the file. It first selects blocks to encrypt from the buffer (Line 10-12). In this step, the number and position of blocks are based on the entropy of the final written buffer. It tries to keep the buffer low-entropy and encrypt more blocks at the same time. After block selection and encryption, the subprocess writes the buffer back to this file (Line 13). If the instruction is another operation such as open or delete, the subprocess will accordingly execute the operation. In this way, each subprocess follows most of the I/O behaviors of benign programs and encrypts the content of victim files.

5. Evaluation

In this section, we evaluate the effectiveness and performance of the imitation-based ransomware attack. Our goal is to understand how this attack works under existing detection tools, and how it is different from conventional ransomware families. Section 5.1 describes our proof-of-concept implementation, and Section 5.2 details our experiment setup. Section 5.3 and Section 5.4 investigates the effectiveness of the attack at different settings. Section 5.5 analyzes the throughput of the attack. Section 5.6 analyzes the robustness of the attack against defense. Section 5.7 provides a case demonstrating how this attack imitates benign programs. In addition, we illustrate the ethical considerations of our evaluation process in Appendix A.4.

5.1. Implementation

We implemented a prototype of the imitation-based attack, referred to as ANIMAGUS. The behavior collection in the benign behavior analysis module is implemented with Windows filesystem minifilter driver framework [37] to collect IRP-level I/O behaviors. This approach can collect fine-grained information and is less likely to be shutdown by ransomware. The whole attack orchestration module is implemented in Rust code and leverages Tokio framework [38] to deal with encryption tasks asynchronously. The approach of encryption and decryption is detailed in Appendix A.1. Regarding cipher, our implementation uses RSA2048 and AES256 with ECB mode for simplicity. We use off-the-shelf cipher implementation [39] instead of rolling a new one. For each victim file, at least 50% of the contents are encrypted. The detailed analysis on its undecryptability is discussed in Appendix A.2. Unlike conventional malware samples, our implementation does not use any anti-analysis tricks or packing techniques because it is for research only and not designed to propagate itself and blackmail others.

5.2. Experiment Setup

Environment. The experiments were conducted on a desktop with a 16-core Intel i5-12600KF CPU (3.69 GHz) and 32GiB of memory. Each ransomware attack runs on a virtual machine that installed Windows 10 and is assigned 4 cores and 8GiB memory. All experiments were performed according to guidelines [40] for malware experiments.

Imitated Target. We choose ten widely-used benign programs, including web browsers (Microsoft Edge, FireFox and Chrome), document editors (WPS Office and Microsoft Office), file compressor (7Zip and WinRAR), compilers (Golang-go and Rustc), and integrated development environment (Visual Studio), as our imitated targets to demonstrate the adaptability of ANIMAGUS. We run each program for ten minutes and collect their I/O behaviors. The statistics of the exploitable patterns in these programs are shown in TABLE 1. In Section 5.3, we choose FireFox as the default imitated target for ANIMAGUS. We will investigate the impact of target choice to attack effectiveness in section 5.4.

TABLE 2: Features of detection tools used in evaluation.

Detection Tool	Approach	Techniques	Available
Kaspersky Total Security	hybrid	all existing detection techniques	✓
360 Total Security	hybrid	all existing detection techniques	✓
Windows Defender	hybrid	all existing detection techniques	✓
Unveil	rule-based	I/O behavior inspection, pattern matching	✗
Redemption	rule-based	I/O behavior inspection, feature scoring	✗
ShieldFS	ML-based	I/O behavior inspection, crypto primitives detection and file recovery	✓

Detection Tools. We choose six ransomware detection tools to investigate the effectiveness of ANIMAGUS. There are three production-level anti-ransomware softwares (i.e., Kaspersky Total Security [15], 360 Total Security [16] , and Windows Defender [17]) and three state-of-the-art research tools of ransomware detection (i.e., Unveil [8], ShieldFS [9], and Redemption [10]). Their features are presented in TABLE 2. The three production-level tools do not detail their techniques, so we assume they apply all existing detection techniques. Our evaluation uses the latest version of the three production-level tools as of 1st August 2022. The authors of Unveil and Redemption do not release their tools,

so we implemented the two tools from scratch based on their research papers. The authors of ShieldFS provided a virtual machine with ShieldFS installed, and we thus use the virtual machine to conduct experiments.

Ransomware Families. To investigate how our attack is different from conventional ransomware, we collect 232 ransomware samples from public repositories such as theZoo [32], VirusShare [33], and the dataset used in ShieldFS [9]. These samples are identified by VirusTotal API [34] into ten recently-active ransomware families, including WannaCry, Avoslocker, XData, TeslaCrypt, Bitman, Vobfus, Dalexis, Yakes, Koxic and phobos. We ensure that samples used in our experiments are alive and capable of encrypting files on victim systems. The behavior difference between ANIMAGUS and these ransomware samples will be investigated in Section 5.3.

Victim Files. We randomly select 1,000 victim files from Govdocs1 [41]. The file extensions of victim files include `pdf`, `jpg`, `doc`, `xls`, `ppt`, `rtf`, `sql`, `txt`. A ransomware sample tends to encrypt all files with these extensions in the victim system. The impact of the number of victim files on the performance of ANIMAGUS will be investigated in Section 5.4.

5.3. Attack Effectiveness

We evaluated ANIMAGUS against six ransomware detection tools to investigate its effectiveness. FireFox is chosen here as the default imitated target for ANIMAGUS. We will investigate the impact of target choice to attack performance in section 5.4. We placed the collected 1,000 victim files into each virtual machine and then conducted ransomware attacks on these machines. For a fair comparison, we ensure every attack experiment is distributed the same resources (4 cores and 8GiB memory). TABLE 3 is an overall result showing whether a ransomware program can successfully encrypt all files under different ransomware detection tools.

Evaluation Against Detection Tools. As we can see, although the six ransomware detection tools are able to detect and terminate most existing ransomware early on their attacks, ANIMAGUS can still encrypt all victim files without being detected. The reason why ANIMAGUS successful eludes the detection tools is that all the I/O behaviors of ANIMAGUS imitate the collected real-world behaviors of the imitated benign program (i.e., FireFox in this experiment). The major difference is the accessed file paths. In the collected behaviors, FireFox reads and writes a number of cache files and cookies; while in our attack, ANIMAGUS reads and writes a number of victim files acting the same as FireFox. This makes detection tools very difficult to tell the difference between them.

The two rule-based detection tools, i.e., Unveil and Redemption, cannot tell the difference based on their heuristics. Unveil observes the entropy of written buffers to see if a program writes many high-entropy buffers to files. It assumes that ransomware writes many encrypted buffers, which are highly-randomized because of the encryption algorithms' nature. However, as described in Section 4.2, the buffer of each ANIMAGUS's write operation only contain a small part of encrypted blocks and the rest is the original contents. Therefore, the entropy of each written buffer remains low. In addition to entropy, Unveil also customizes a set of I/O sequence patterns to assist its detection. For example, one pattern is that ransomware reads a low-entropy buffer from a file, then writes a high-entropy buffer to a new file, and finally deletes the original file. However, the I/O sequence of ANIMAGUS is the same as the imitated benign program. Therefore, Unveil cannot detect ANIMAGUS based on the entropy of written buffers and its custom I/O sequence patterns. Redemption, on the other hand, calculates a malice score for each process based on its custom features. The features include entropy of written buffers, length of overwritten buffers, number of delete operations, number of directories traversed, types of accessed files, and access frequency. Redemption empirically assigns weights to these features and calculates scores for processes based on the weights. If the score of a process exceeds a custom threshold (i.e., 0.12 in its paper), then the process is determined to be malicious. In terms of all these features except for the accessed file types, ANIMAGUS behaves similarly to the imitated benign program. However, the feature of accessed file types was assigned a relatively low weight in Redemption to reduce the false positive rate because benign programs such as document editors and media players also need to access a wide range of file formats. As a result, the malice score of ANIMAGUS is similar to that of FireFox, which is far less than its threshold.

ShieldFS, a ML-based detection tool, also fails to detect ANIMAGUS. It trains a random forest model based on its custom features to discriminate malicious behaviors. The features used include the number of traversed directories, the number of read files, the number of written files, the number of renamed files, the number of accessed files' types, and the average entropy of written buffers. We cannot directly investigate why ShieldFS did not detect ANIMAGUS because the authors did not release its kernel driver nor its trained model. Fortunately, the authors provided the raw training dataset, which contains 185,140,123 IRPs of 383 ransomware samples and 2,245 benign programs. We trained a decision tree [42] model based on the dataset and regarded the model as an alternative to ShieldFS to facilitate our

TABLE 3: Attack result of ANIMAGUS and traditional ransomware against existing detection tools. ANIMAGUS without imitation is referred as ANIMAGUS^{-}.

Program	v.s. Detection Tools					
	Kaspersky	360	Defender	Unveil	Redemption	ShieldFS
ANIMAGUS	✓	✓	✓	✓	✓	✓
ANIMAGUS^{-}	✗	✗	✗	✓	✗	✗
WannaCry	✗*	✗*	✗*	✗	✗	✗
Avoslocker	✗*	✗*	✓	✓	✓	✗
XData	✗*	✗*	✗*	✓	✓	✗
TeslaCrypt	✗*	✗*	✗*	✗	✗	✗
Bitman	✗*	✗*	✗*	✗	✗	✗
Vobfus	✗*	✗*	✗*	✓	✓	✗
Dalexis	✗*	✗*	✗*	✗	✓	✗
Yakes	✗*	✗*	✗*	✗	✗	✗
Koxic	✗*	✗*	✗	✓	✓	✗
phobos	✗*	✗*	✗*	✓	✗	✗

✓: attack successfully; ✗: be detected; ∗: be detected as soon as it started.

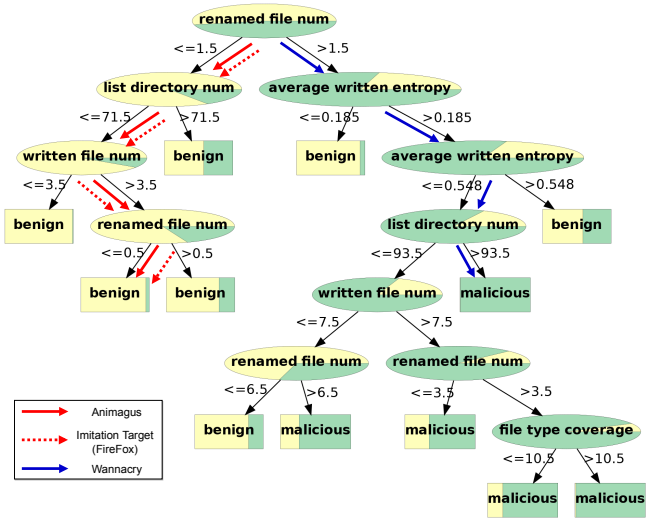


Figure 6: Decision tree trained with ShieldFS's raw data. The color of each node is corresponding to the training data passed through the node, yellow denotes benign programs, green denotes ransomware samples. Arrows denote the decision paths of the tree processing the IRPs of each program.

in-depth analysis. The way we calculate feature values for each program in the dataset is consistent with ShieldFS. The feature values of each program are normalized by the number of IRPs. We used 80% of the data for training and 20% for testing. After being trained, the model reports 0.9891 F1-score on testing set. Figure 6 is a visualization of shallow layers of the trained decision tree. The node denotes the feature used to make the decision, and the edge denotes the decision threshold of the feature. As we can see, it treats the number of renamed files, the number of directories traversed, and the average written entropy as more important criteria for decision making. In terms of these features, ANIMAGUS behaves similarly to the imitated benign program. Therefore, as the red arrow denotes, the tree considers ANIMAGUS as a benign program. As a result, although the ML-based approach can identify the difference between benign programs and conventional ransomware samples, ShieldFS cannot detect ANIMAGUS.

The three production-level detection tools, i.e., Kaspersky Total Security, 360 Total Security, and Windows Defender, also fail to detect ANIMAGUS. We cannot profoundly analyze how they detect ransomware because they did not detail their techniques. According to our experiments, the three tools are good at static and sandbox analysis. Before being executed, all the ransomware samples used in our evaluation are quickly detected by the three tools based on the malware signatures. We can only manually put these samples into the whitelists of the tools to make them executable. However, the samples except for Avoslocker were terminated as soon as they started. This is probably because the tools perform sandbox analysis before executing the samples. In terms of static analysis, the signature of ANIMAGUS is not recorded in their databases, and thus the tools cannot detect ANIMAGUS statically. In terms of dynamic analysis, the three tools may take a similar strategy to the research tools described above. As a result, they are unable to detect ANIMAGUS.

Comparison with Conventional Ransomware. We collected 232 samples of ten recently-active ransomware families to investigate how ANIMAGUS is different from them. We randomly choose one sample from each family. This is because samples of a family share an almost identical behavioral pattern at runtime as demonstrated in Appendix B.4. As we can see from TABLE 3, most of them were detected by existing detection tools, while ANIMAGUS was not detected by any of the tools. We take ShieldFS as an example to show the difference between conventional ransomware and ANIMAGUS from a defender's point of view. As the blue arrow in Figure 6 demonstrates, the WannaCry, one of the most famous ransomware families, was quickly detected based on the number of renamed files, the number of directories traversed, and the average written entropy. The reason is that conventional ransomware samples tend to greedily encrypt files and write a large number of high-entropy buffers to files. ANIMAGUS, on the contrary, encrypts files sporadically, disguised by normal behaviors. Compared to conventional ransomware, ANIMAGUS has a higher chance of success, although it takes more time to encrypt files.

Attack Without Imitation. We also conduct an ablation study to investigate whether imitation improves the attack's effectiveness. We remove the imitation and orchestration procedure from ANIMAGUS to make a new version, referred to as ANIMAGUS$^-$. ANIMAGUS$^-$ keeps the same encryption approach but randomly schedules encryption tasks and spawns subprocesses for the tasks. As we can see from TABLE 3, ANIMAGUS$^-$ was still detected by most of the existing detection tools. There are several reasons why ANIMAGUS$^-$ is easy to detect. First, the access frequency of ANIMAGUS$^-$ is different from benign programs. ANIMAGUS$^-$ tends to spawn a few subprocesses and rapidly encrypt files. Therefore, its access frequency is much higher than benign programs. Second, all I/O behaviors of ANIMAGUS$^-$ are for encryption. ANIMAGUS$^-$ would only target victim files, while ANIMAGUS would also perform some operations to "dummy" files in addition to victim files. The comparison between ANIMAGUS and ANIMAGUS$^-$ shows that imitating benign programs helps ANIMAGUS disguise its encryption behaviors.

5.4. Impact of Imitated Targets

We investigated if existing detection tools can detect ANIMAGUS that imitates different targets. The chosen targets are MS Edge, FireFox, Chrome, WPS Office, MS Office, 7Zip, WinRAR, Golang-go, Rustc, and Visual Studio. First, we collected 6,000 runtime records of benign behaviors by running each program with real-world workloads, e.g., using browsers to randomly visit websites, using document editors to randomly open and edit some files, or using file compressors to pack and unpack archives. Next, we used the six detection tools to detect the collected benign behaviors, the ransomware samples, and ANIMAGUS with

different imitated targets. TABLE 4 shows the quantitative detection results. Note that we cannot run Visual Studio, Rustc, and Golang-go in the ShieldFS virtual machine, so we omit them. From the table we can see, these tools achieve quite low FNR when detecting ransomware. In contrast, they get quite high FNR when detecting ANIMAGUS. This result demonstrates that most tools successfully detect traditional ransomware samples but cannot effectively detect most variants of ANIMAGUS. In particular, ANIMAGUS^{WinRAR} can be detected by Unveil and Redemption. This is because the imitated behavior of WinRAR contains a fragment that rapidly writes many high-entropy files. ANIMAGUS imitates the behavior and gets detected. This shows that the imitated target slightly impacts the effectiveness of ANIMAGUS. Besides, the imitated target also impacts to attack throughput as we present in Appendix B.1.

TABLE 4: Detection results of different detectors against ANIMAGUS and traditional ransomware samples.

	FPR	v.s. ransomware FNR	v.s. ANIMAGUS FNR
Kaspersky	0.0%	0.0%	100.0%
360	0.0%	0.0%	100.0%
Defender	0.0%	10.0%	100.0%
Unveil	0.2%	50.0%	80.0%
Redemption	15.6%	50.0%	90.0%
ShieldFS	0.0%	0.0%	100.0%

5.5. Attack Throughput

One main disadvantage of ANIMAGUS is the time spent in encryption. In order to disguise its encryption behaviors, it is necessary to follow the same pattern as the imitated benign program. Therefore, the time spent in encryption depends on the exploitable file placeholders, i.e., accessed files conforming to the three exploitable patterns in the behavior template. The more encryption tasks in the behavior template, the more files ANIMAGUS can encrypt within a certain time slot. Although ANIMAGUS controls subprocesses to speed up the imitated behaviors, it is still necessary to evaluate its overall throughput.

Figure 7 shows the comparison of encryption time between traditional ransomware and ANIMAGUS. The ransomware samples used here are in line with Section 5.3. The ANIMAGUS used here are ten different versions, each imitating a specific benign program (in line with Section 5.4). The individual encryption times of ransomware samples and ANIMAGUS are shown in TABLE 8 and TABLE 7 in Appendix B.1, respectively. On average, Animagus took -17.21%, 10.45%, 55.30%, 46.42%, and 33.39% more time to encrypt 200, 400, 600, 800, and $1,000$ files, respectively, than traditional ransomware samples. Although ANIMAGUS is slower, it is much more difficult to detect. According to Figure 4, existing detection tools can easily detect these ransomware samples; In contrast, detection tools get high false negative rates ($80\% - 100\%$) when facing ANIMAGUS. Therefore, the encryption time of ANIMAGUS is not much longer than that of traditional ransomware, but the attack success rate is much higher.

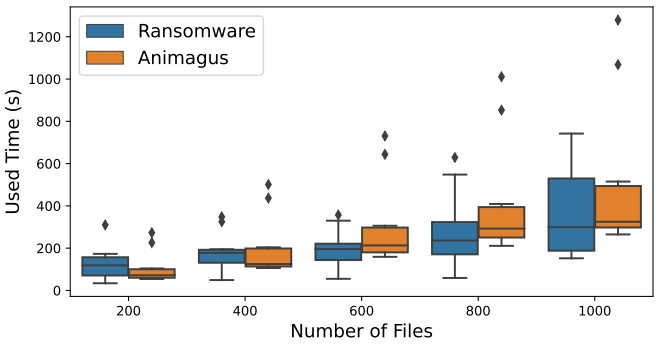


Figure 7: Time taken by ANIMAGUS and existing ransomware samples to encrypt different numbers of files.

5.6. Robustness Against Defense

It is important to investigate if one can construct a simple rule to distinguish ANIMAGUS. By digging into the implementation of ANIMAGUS, defenders may discover some different behaviors between ANIMAGUS and imitated programs. There are three fundamental differences, i.e., the file types accessed, the scanning behavior, and dummy reads and writes. We investigate if ANIMAGUS behaves much different from benign programs.

We implemented detectors based on the three rules. To evaluate the detectors, we collected 6,000 runtime records of benign behaviors. The records are the same as the ones used in Section 5.4. In addition, we also collected the runtime records of different ANIMAGUS, each imitating one of these benign programs. We oversampled the records of ANIMAGUS to make the dataset balanced. We fine-tuned thresholds of the detectors using these collected runtime records. During the fine-tuning process, we consider a file to be high-value if it is one of office files (e.g., `doc`, `xls`, `ppt`, and `pdf`), program files (e.g., `cpp`, `go`, `js`, and `rs`), picture files (e.g., `jpg` and `png`), or database files (e.g., `db`, `sqlite`, and `sql`). TABLE 5 presents the best F1-score results of the detectors and corresponding TPR and FPR. Figure 8 is the ROC curve of each detector. Overall, they cannot effectively detect ANIMAGUS without incurring a considerable FPR. Below we detail the feature engineering of each detector and why they are ineffective.

TABLE 5: Detection results of three new detectors against ANIMAGUS and benign programs. Presented are the best results of fine-tuning each detector according to F1-score.

	TPR	FPR	F1-score
file-type based detector	0.900	0.499	0.751
static scanning based detector	1.000	0.338	0.856
temporal scanning based detector	0.800	0.051	0.865
dummy-access based detector	0.600	0.966	0.468

In terms of the file-type based detector, we investigated three file-type based metrics, i.e., the average number of write/delete/rename operations on high-value files, the peak frequency of write/delete/rename operations on high-value files, and the average number of unique high-value files

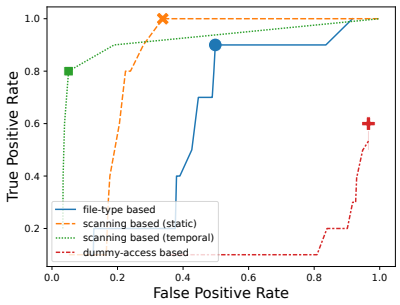


Figure 8: ROC of three detectors. The marker of each curve denotes the best F1-score result.

written/deleted/renamed. We analyzed each metric on the collected runtime records. Statistics are present in Figure 10 in Appendix B.2. From Figure 10 we can see, none of the metrics can distinguish ANIMAGUS. This is because many benign programs would also access file types commonly targeted by ransomware. For example, browsers would frequently write data to many database files; Editors would write contents to text or program files; Compressors would delete the original files if the delete-after-compression flag is enabled; Rustc, in particular, would rapidly write data to more than 500 .rs files when downloading external crates. On the other hand, ANIMAGUS not only accesses high-value files, but also performs dummy reads and writes to non-high-value files. Figure 12 in Appendix B.2 demonstrates the percentage of accessed high-value files out of all accessed files for each program. We can see that 42.44% of files that ANIMAGUS accesses are high-value, which cannot distinguish ANIMAGUS. After feature engineering, we implemented a detector based on the number of write/delete/rename on high-value files and fine-tuned its threshold. With its best F1-score result, it gets 0.900 TPR and 0.499 FPR.

In terms of the scanning based detector, we investigated three scanning based metrics, i.e., the average number of scanning operations, the peak frequency of scanning operations, and the average number of unique folders scanned. We analyzed each metric on the collected runtime records and statistics are present in Figure 11 in Appendix B.2. Overall, none of the metrics can distinguish ANIMAGUS. This is because many benign programs would also scan a large amount of folders. For example, compressors would rapidly traverse folders that need to be packed; Visual Studio would rapidly traverse the folders of target projects and folders containing external libraries or `.h` files; MS Office would scan folders containing fonts, caches, or metadata. On the other hand, ANIMAGUS only scans a few folders to target a certain number of victim files in each cycle (see Line 14 in Algorithm 1), which makes it difficult to detect. In addition to static scanning metrics, we also investigate if ANIMAGUS can be detected by a temporal scanning feature. Figure 12 in Appendix B.2 presents scanning frequency in the first ten seconds of different programs. We can see that many benign programs would also scan many folders in their initial phase. After feature engineering, we used the peak frequency of scanning operations as the rule of the static scanning based detector, and used the percentage of

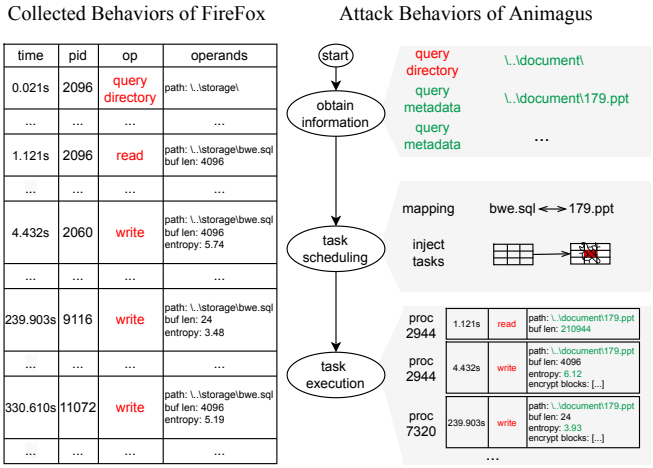


Figure 9: Imitation process of ANIMAGUS^{FireFox}. The data used here is from the actual collected IRPs. The behavior difference is highlighted in green.

scanning operations in the initialization phase to all scanning operations as temporal the scanning based detector. We fine-tuned their thresholds. With their best F1-score results, the static detector achieves 1.000 TPR and 0.338 FPR and the temporal detector achieves 0.800 TPR and 0.051 FPR.

The dummy-access based detector is difficult to implement. One reasonable observation is that ANIMAGUS would not perform dummy access to high-value files. Besides, the access pattern of dummy files may not be one of the three exploitable patterns because ANIMAGUS would leverage as many exploitable files as possible to schedule encryption tasks. So one reasonable metric is the percentage of files accessed in exploitable patterns among all accessed non-high-value files throughout the lifetime of each program. Figure 14 in Appendix B.2 presents the results. From the figure we can see, the average percentage is between 3.92% to 28.14% for benign programs, and 25.23% for ANIMAGUS. After analysis, we implemented a corresponding detector and fine-tuned its threshold. With its best F1-score result, it gets 0.600 TPR and 0.966 FPR.

In summary, constructing a practical detection rules to distinguish ANIMAGUS is non-trivial. Defenders must thoroughly investigate the runtime behaviors of ANIMAGUS and perform heavy feature engineering to produce a useful rule.

5.7. Case Study

We present a case study on how ANIMAGUS exploits behaviors of benign programs. Figure 9 illustrates the imitation process of ANIMAGUS^{FireFox}. During the offline preparation phase, it analyzes the collected behaviors and identifies exploitable files. In this example, the access to `bwe.sql` follows the Read-Write exploitable pattern. During the online attack phase, the program first follows the same behavior as FireFox to query directory information from the victim system. Next, it queries the metadata of each file in the directory. Taking the `179.ppt` as an example, the program will know the size of this file from the metadata.

After that, it starts scheduling encryption tasks. It maps an exploitable file, i.e., the `bwe.sql` from the behaviors of the imitation target, to the victim file, and injects encryption tasks to the access sequence of the exploitable file. The tasks are injected into some write operations of the access sequence based on the number of write operations and the file size. Finally, it assigns the tasks to subprocesses, and the subprocesses access files on schedule.

As we can see from the figure, there are four main differences in behaviors between ANIMAGUS and the program it imitates. The first is the file path of each operation. Every ANIMAGUS access operates on the actual file paths on the victim system. The second is the additional operation of query metadata. Once a list of files in a directory is obtained, ANIMAGUS queries the metadata of these files. This is because it needs to know the size of each file in order to schedule encryption tasks. Note that most programs would also query metadata of files after obtaining directory information, so this cannot be a feature to identify ANIMAGUS. The third is the length of read buffers. When encountering an operation that reads the file for the first time, ANIMAGUS reads the entire contents of the file. This is because it needs to know the content before encrypting. The fourth is the entropy of written buffers. The entropy inevitably differs from the imitated behavior because the written contents are different. ANIMAGUS tends to write low-entropy buffers to disguise its encryption behavior. Except for the four differences, ANIMAGUS behaves the same as the program.

6. Discussion

6.1. Countermeasures

In Section 5, we illustrate that it is difficult to discriminate imitation-based ransomware attack if one only considers I/O behaviors. Here we propose some potential strategies to counter the attack.

Combine Multiple Techniques. In addition to access to filesystem, a ransomware program would also connect to its C&C server and/or performs encryption calculation. Combining the analysis of the network activities and the cpu/memory usage could offer a more precise detection result. There are a few studies in this direction. For example, Ransomspector [14] notices that many existing ransomware samples tend to send packets to a large amount of different hosts. It thus considers this behavior as a detection pattern. RWGuard [11] places hooks at the beginning of the CryptoAPI library functions to monitor the invocation activities. PayBreak [21], similar to RWGuard, places hooks to CryptoAPI in order to obtain encryption keys. These ad-hoc mitigations cannot make ransomware infeasible. ANIMAGUS neither sends packets to different hosts, nor does it rely on CryptoAPI. However, combining all aspects of these activities for hybrid analysis is promising. For example, one can trace the real-time memory of a suspicious program to see if it performs many encryption operations to a buffer read from filesystem. This fine-grained analysis can counter ANIMAGUS although it introduces considerable overhead.

Data Backup. There are many off-the-shelf backup software options, such as OneDrive and Google Drive. Unfortunately, recent studies [43] show that a small part of ransomware can obtain OS kernel privilege to terminate or destroy these backup programs. There are also some studies [43], [44], [45], [46] try to develop a hardware-assisted backup to fight against ransomware. Their key insight is to isolate backup data from operating system, so that ransomware cannot destroy backup data. However, they could also introduce considerable time overhead or space overhead without a solid detection technique. Backing up files once a file is accessed suspiciously is a promising strategy to counter the imitation-based ransomware attack.

6.2. Potential Benefit

Existing detection tools can leverage different versions of ANIMAGUS as adversarial examples to improve their detection. For example, Redemption can adjust the weights of its features by studying ANIMAGUS IRP logs; ShieldFS can take the logs as training data to retrain its model. Besides, recently, several evaluation frameworks [47], [48] were proposed to test the effectiveness of ransomware detectors. These frameworks can also integrate ANIMAGUS as an evaluation metric for a more thorough evaluation.

7. Conclusion

In this paper, we aim to explore the limits of ransomware detection techniques that leverage I/O behaviors to discriminate malicious programs. To this end, we propose an imitation-based ransomware attack that imitates benign programs' behaviors and disguises encryption tasks. It first abstracts a behavior template from real-world benign I/O behaviors. During online attacks, it acts in the same behavior pattern as the behavior template and encrypts files. Our experimental results show that although current detection tools can identify malicious behaviors of existing ransomware samples, our attack can still successfully elude the tools. We investigate in detail why the detection techniques are ineffective and how ANIMAGUS is different from existing ransomware samples. We highlight that the I/O-based detection alone might not be sufficient to identify ransomware and should be complemented by other analyses to achieve better detection accuracies.

8. Acknowledgement

We thank the anonymous reviewers for their insightful feedback. We also thank the authors of ShieldFS for their valuable dataset. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No.62022046, 92167101, U1911401, 62021002).

References

[1] D. Braue, "Global ransomware damage costs predicted to exceed $265 billion by 2031." https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-250-billion-usd-by-2031/, Cybercrime Magazine, 2022, (Online; visited on July 13, 2022).

[2] M. Eddy and N. Perlroth, "Cyber attack suspected in german woman's death." https://www.nytimes.com/2020/09/18/world/europe/cyber-attack-germany-ransomeware-death.html, The New York Times, 2020, (Online; visited on July 13, 2022).

[3] K. Cabaj and W. Mazurczyk, "Using software-defined networking for ransomware mitigation: The case of cryptowall," *IEEE Netw.*, vol. 30, no. 6, pp. 14–20, 2016.

[4] M. Keshavarzi and H. R. Ghaffary, "I2ce3: A dedicated and separated attack chain for ransomware offenses as the most infamous cyber extortion," *Computer Science Review*, vol. 36, p. 100233, 2020.

[5] H. Oz, A. Aris, A. Levi, and A. S. Uluagac, "A survey on ransomware: Evolution, taxonomy, and defense solutions," *ACM Computing Surveys (CSUR)*, 2021.

[6] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the gordian knot: A look under the hood of ransomware attacks," in *International conference on detection of intrusions and malware, and vulnerability assessment (DIMVA)*. Springer, 2015, pp. 3–24.

[7] D. Y. Huang, M. M. Aliapoulios, V. G. Li, L. Invernizzi, E. Bursztein, K. McRoberts, J. Levin, K. Levchenko, A. C. Snoeren, and D. McCoy, "Tracking ransomware end-to-end," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 618–631.

[8] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 757–772.

[9] A. Continella, A. Guagnelli, G. Zingaro, G. D. Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "Shieldfs: a self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 2016, pp. 336–347.

[10] A. Kharraz and E. Kirda, "Redemption: Real-time protection against ransomware at end-hosts," in *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, ser. Lecture Notes in Computer Science, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds., vol. 10453. Springer, 2017, pp. 98–119.

[11] S. Mehnaz, A. Mudgerikar, and E. Bertino, "Rwguard: A real-time detection system against cryptographic ransomware," in *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, ser. Lecture Notes in Computer Science, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., vol. 11050. Springer, 2018, pp. 114–136.

[12] M. A. Ayub, A. Continella, and A. Siraj, "An i/o request packet (irp) driven effective ransomware detection scheme using artificial neural network," in *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE, 2020, pp. 319–324.

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

[14] F. Tang, B. Ma, J. Li, F. Zhang, J. Su, and J. Ma, "Ransomspector: An introspection-based approach to detect crypto ransomware," *Comput. Secur.*, vol. 97, p. 101997, 2020.

[15] "Kaspersky: cybersecurity that's always a step ahead." https://usa.kaspersky.com/, Kaspersky Home Page, 2022, (Online; visited on July 13, 2022).

[16] "360 total security," https://www.360totalsecurity.com/, 360 Security Home Page, 2022, (Online; visited on July 13, 2022).

[17] "Windows security: Defender, antivirus and more for windows 11." https://www.microsoft.com/en-us/windows/comprehensive-security, Microsoft, 2022, (Online; visited on July 13, 2022).

[18] B. A. S. Al-rimy, M. A. Maarof, and S. Z. M. Shaid, "Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions," *Comput. Secur.*, vol. 74, pp. 144–166, 2018.

[19] F. Cicala and E. Bertino, "Analysis of encryption key generation in modern crypto ransomware," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 2, pp. 1239–1253, 2022.

[20] L. Zhang-Kennedy, H. Assal, J. N. Rocheleau, R. Mohamed, K. Baig, and S. Chiasson, "The aftermath of a crypto-ransomware attack at a large academic institution," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 1061–1078.

[21] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "Paybreak: Defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, Eds. ACM, 2017, pp. 599–611.

[22] L. Invernizzi, S. Miskovic, R. Torres, C. Kruegel, S. Saha, G. Vigna, S. Lee, and M. Mellia, "Nazca: Detecting malware distribution in large-scale networks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[23] C. Lever, P. Kotzias, D. Balzarotti, J. Caballero, and M. Antonakakis, "A lustrum of malware network communication: Evolution and insights," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 788–804.

[24] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 161–175.

[25] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Z. Snow, F. Monrose, and M. Antonakakis, "The circle of life: A large-scale study of the iot malware lifecycle," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 3505–3522.

[26] Y. Xu, Z. Yin, Y. Hou, J. Liu, and Y. Jiang, "MIDAS: safeguarding iot devices against malware via real-time behavior auditing," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4373–4384, 2022.

[27] Z. Yin, Y. Xu, C. Zhou, and Y. Jiang, "Empirical study of system resources abused by iot attackers," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 39:1–39:13.

[28] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features," in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, A. Birrell and E. G. Sirer, Eds. USENIX Association, 2013, pp. 187–198.

[29] M. Brengel and C. Rossow, "YARIX: scalable yara-based malware intelligence," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 3541–3558.

[30] O. E. David and N. S. Netanyahu, "Deepsign: Deep learning for automatic malware signature generation and classification," in *2015 International Joint Conference on Neural Networks, IJCNN 2015, Killarney, Ireland, July 12-17, 2015*. IEEE, 2015, pp. 1–8.

[31] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*. IEEE Computer Society, 2007, pp. 421–430.

[32] "thezoo - a live malware repository." https://github.com/ytisf/theZoo, ytisf, 2022, (Online; visited on July 13, 2022).

[33] "Virusshare.com - because sharing is caring." https://virusshare.com/, VirusShare, 2022, (Online; visited on July 13, 2022).

[34] "Virustotal developer hub." https://developers.virustotal.com/, VirusTotal Team, 2022, (Online; visited on July 13, 2022).

[35] "Process monitor v3.90." https://docs.microsoft.com/en-us/sysinternals/downloads/procmon, Microsoft, 2022, (Online; visited on July 13, 2022).

[36] "strace: linux syscall tracer." https://strace.io/, strace, 2022, (Online; visited on July 13, 2022).

[37] "Filter manager concepts." https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts, Microsoft, 2022, (Online; visited on July 13, 2022).

[38] "Tokio - an asynchronous rust runtime." https://tokio.rs/, Tokio, 2022, (Online; visited on July 13, 2022).

[39] "Collection of block cipher algorithms written in pure rust." https://github.com/RustCrypto/block-ciphers, RustCrypto, 2022, (Online; visited on July 13, 2022).

[40] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen, "Prudent practices for designing malware experiments: Status quo and outlook," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 65–79.

[41] S. L. Garfinkel, P. F. F. Jr., V. Roussev, and G. W. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *Digit. Investig.*, vol. 6, no. Supplement, pp. S2–S11, 2009.

[42] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. Routledge, 2017.

[43] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, "Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2231–2244.

[44] X. Wang, Y. Yuan, Y. Zhou, C. C. Coats, and J. Huang, "Project almanac: A time-traveling solid-state drive," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse, and C. Fetzer, Eds. ACM, 2019, pp. 13:1–13:16.

[45] J. Park, Y. Jung, J. Won, M. Kang, S. Lee, and J. Kim, "Ransomblocker: a low-overhead ransomware-proof SSD," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 34.

[46] B. Reidys, P. Liu, and J. Huang, "Rssd: Defend against ransomware with hardware-isolated network-storage codesign and post-attack analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022, Lausanne, Switzerland*. ACM, 2022.

[47] J. Han, Z. Lin, and D. E. Porter, "On the effectiveness of behavior-based ransomware detection," in *Security and Privacy in Communication Networks - 16th EAI International Conference, SecureComm 2020, Washington, DC, USA, October 21-23, 2020, Proceedings, Part II*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, N. Park, K. Sun, S. Foresti, K. R. B. Butler, and N. Saxena, Eds., vol. 336. Springer, 2020, pp. 120–140.

[48] A. Gupta, A. Prakash, and N. Scaife, "Prognosis negative: Evaluating real-time behavioral ransomware detectors," in *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 353–368.

Appendix A. Implementation Details

A.1. Encryption and Decryption

We leverage a hybrid cryptosystem to encrypt files similar to existing ransomware families. The cryptosystem includes a asymmetric key generation algorithm $\mathcal{K}$, an encryption algorithm $\mathcal{E}$ and a decryption algorithm $\mathcal{D}$. First, our program generates a pair of asymmetric public-private key on the C&C server:

$$(priv\_k, pub\_k) \leftarrow \mathcal{K} \tag{1}$$

Next, during attacks, the program generates a unique session key s_k for each file, and encrypts the content M of each file using the session key:

$$C1 = \mathcal{E}_{s\_k}(M) \tag{2}$$

In the end, every session key is encrypted with the public key and left together with the encrypted file contents:

$$C2 = \mathcal{E}_{pub\_k}(s\_k) \tag{3}$$

The approach to decrypt a file is straight-forward. With the private key, the victim can first decrypt the chunk of encryption information appended to each encrypted file:

$$s\_k = \mathcal{D}_{priv\_k}(C2) \tag{4}$$

Finally, our program uses the session key to decrypt contents. In this way, it can recover all the locked files.

$$M = \mathcal{D}_{s\_k}(C1) \tag{5}$$

After a file is encrypted, a chunk of encryption information is appended to the file's content as Figure 5 demonstrates. The encryption information indicates which blocks are encrypted, which cipher we choose, and what the encrypted session key is. There are a variety of options for the cipher and the cipher mode. The cipher can be any combination of a symmetric block cipher (such as AES256, Blowfish and 3DES) and an asymmetric cipher (such as RSA and DSA). The cipher mode can be ECB, CTR, CBC, etc. We use RSA2048 and AES256 with ECB mode in our prototype for the sake of simplicity.

A.2. Intermittent Encryption

Encrypting intermittent blocks is our indispensable design. Without this strategy, multiple processes are unlikely to synergistically encrypt the same file and control the entropy of each written buffer. Even so, there is still no way for victims to decrypt locked files without a decryption key. First, the blocks are selected randomly, and the victims thus cannot know which parts are encrypted. Second, the blocks are impossible to decrypt because of the nature of modern ciphers. In the implementation of ANIMAGUS, we ensure that at least 50% of the file contents are encrypted. Therefore, for any file whose size is larger than 32 bytes, at

least 128 Bit of its content will be encrypted by ANIMAGUS. Even though the victim knows which bits are encrypted, it will take at least 10^{18} years to recover the content.

TABLE 6: Recoverability of `.mov` files with different sizes against different encryption rates.

	0.01%	0.05%	0.10%	0.50%	1.00%
1M	○	◑	◑	●	●
5M	○	◑	◑	●	●
10M	○	◑	◑	●	●
30M	○	◑	◑	●	●
90M	○	◑	◑	●	●
150M	○	○	◑	●	●
210M	○	○	○	●	●
270M	○	○	○	◑	●

○: playable; ◑: not playable but recoverable; ●: unrecoverable.

One may argue that some media files may have highly redundant data, which makes encrypted files recoverable. We conducted experiments on `.mov` files to investigate if the encryption strategy can destroy media files. We choose `.mov` because it is one of the most widely-used media formats. After a file is encrypted, we measure if it is playable and recoverable. The video player used here is QuickTime Player, and the recovery tool is Wondershare Recoverit. From TABLE 6 we can see, when the encryption rate is 0.10%, the encrypted file is recoverable. However, when the encryption rate is 1.00%, the encrypted file is unlikely to be recovered. Therefore, encrypting 50% of a file is enough to destroy it.

A.3. Speed Up Replay

Algorithm 3 illustrates our speedup strategy during execution task scheduling. Users of ANIMAGUS preset a throughput control variable T before attacks. During attacks, this speedup algorithm receives a list of encryption instructions from the encryption task scheduling module. Each instruction contains information which instructs a specific subprocess when and how to access a specific file. By analyzing these instructions, we can know the number of tasks and total encryption time contained in the instruction list, and thus calculate a speedup rate using the throughput control variable (Line 1-3). Next, we can modify the time of each instruction according to the speedup rate and finally speed up the whole attack process (Line 4-6).

Algorithm 3: Speed Up Replay During Scheduling

Input : A list of encryption instructions $Instrs$
Throughput control variable T

1 *taskNum* ← analyzeTask($Instrs$)
2 *totalTime* ← analyzeTimeRange($Instrs$)
3 *speedUpRate* ← $T \times \frac{totalTime}{taskNum}$
4 **for** *Instr* **in** *Instrs* **do**
5 | *Instr.time* ← *Instr.time / speedUpRate*
6 **end**

A.4. Ethical Considerations

We consider ethics as a top priority when conducting experiments. All experiments were conducted in a virtual environment and did not affect any real-world end-users. The implemented prototype of the imitation-based attack does not contain any functionalities of infection or propagation, so it will not do harm to the public. We believe that the experiment setup poses no ethical issues and is sufficient to investigate the limits of I/O based ransomware detectors.

We are engaging in disclosure procedures with three production-level detection providers, i.e., Kaspersky [15], 360 [16], and Windows Defender [17]. They have received our prototype and are actively refining their strategies. Note that production-level detectors also combine other malware detection techniques, e.g., infection detection and static signature matching, to prevent large-scale spread of malware. The idea of this paper only challenges the I/O based dynamic analysis of these detectors. Therefore, as long as other malware detection techniques are effective, even inspired by this paper, cybercriminals cannot construct ransomware to evade these production-level detectors.

For the interest of academia, we will release the prototype in a binary format. We make sure that the binary is only usable in a specific virtual environment and cannot be reused as a submodule of any real-world attacks. Besides, we will release the prototype only when the detection providers can 100% defend against the imitation-based attack.

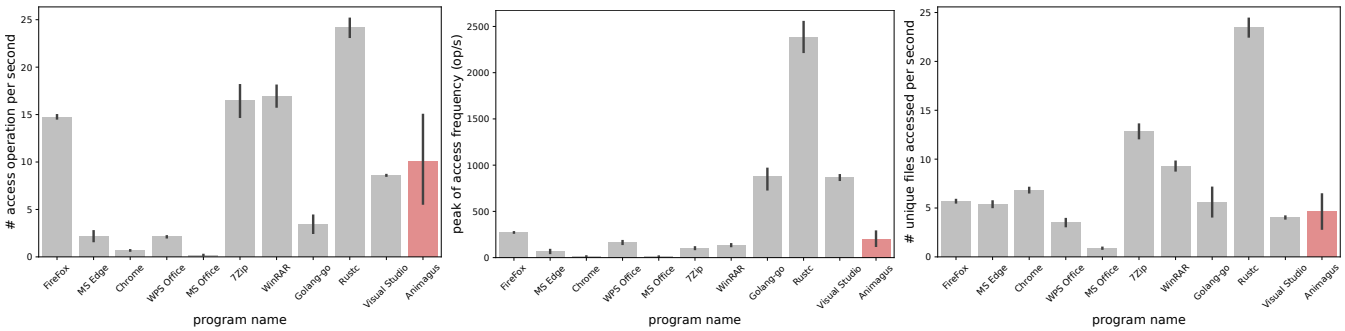
Appendix B. Additional Experiment Results

B.1. Throughput of Individual Samples

We analyzed individual throughput of ANIMAGUS and ransomware families. The results are shown in TABLE 7 and TABLE 8. From TABLE 7 we can see, the attack throughput of ANIMAGUS depends on imitated targets. The key factor is the number of encryption tasks in each behavior template. Behavior templates are extracted from runtime behaviors of imitated targets. Some programs provide templates with many encryption tasks, some do not. Users can adjust attack time by choosing a suitable behavior template.

TABLE 7: Time spent by different versions of ANIMAGUS in encrypting different numbers of files.

	200 files	400 files	600 files	800 files	1000 files
ANIMAGUS^{FireFox}	56s	112s	193s	254s	320s
ANIMAGUS$^{MS\ Edge}$	226s	437s	644s	853s	1068s
ANIMAGUS^{Chrome}	273s	501s	731s	1011s	1279s
ANIMAGUS$^{WPS\ Office}$	81s	110s	233s	323s	330s
ANIMAGUS$^{MS\ Office}$	134s	234s	334s	397s	512s
ANIMAGUS^{7Zip}	63s	125s	178s	262s	295s
ANIMAGUS^{WinRAR}	54s	106s	159s	211s	265s
ANIMAGUS$^{Golang-go}$	59s	117s	173s	232s	289s
ANIMAGUS^{Rustc}	63s	125s	186s	249s	306s
ANIMAGUS$^{Visual\ Studio}$	89s	182s	272s	350s	431s

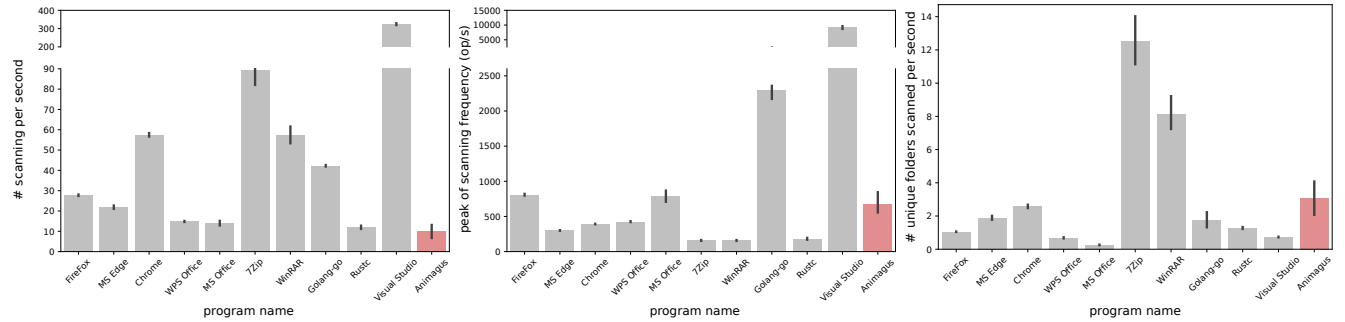


(a) The number of write/delete/rename operations on the files with high-value types over program lifetime (normalized by time).

(b) The peak frequency of write/delete/rename operations on the files with high-value types throughout program lifetime.

(c) The number of unique high-value files written/deleted/renamed over program lifetime (normalized by time).

Figure 10: Statistics of collected IRPs of different programs on three file-type based detection strategies.



(a) The number of scanning operations over program lifetime (normalized by time).

(b) The peak frequency of scanning operations throughout program lifetime.

(c) The number of unique folders scanned over program lifetime (normalized by time).

Figure 11: Statistics of collected IRPs of different programs on three scanning based detection strategies.

TABLE 8: Time spent by ransomware samples in encrypting different numbers of files.

	200 files	400 files	600 files	800 files	1000 files
WannaCry	58s	152s	195s	289s	301s
Avoslocker	34s	49s	55s	59s	161s
XData	159s	195s	226s	335s	496s
TeslaCrypt	151s	178s	196s	203s	541s
Bitman	129s	325s	330s	629s	742s
Vobfus	58s	96s	101s	160s	175s
Dalexis	109s	124s	131s	142s	152s
Yakes	173s	184s	207s	219s	227s
Koxic	109s	177s	182s	253s	297s
phobos	310s	348s	357s	548s	730s

B.2. Feature Engineering of New detectors

File-type based detector. We investigated three file-type based metrics, i.e., the average number of write/delete/rename operations on high-value files over program lifetime, the peak frequency of write/delete/rename operations on high-value files over program lifetime, and the average number of unique high-value files written/deleted/renamed over program lifetime. We analyzed each metric on collected runtime records. Figure 10 presents the overall statistics. In Figure 10a, the average number of write/delete/rename operations per second is between 0.20 to 24.15 for benign programs, and 10.08 for ANIMAGUS. In Figure 10b, the average peak frequency of write/delete/rename operations is between 11.50 to 2386.00 for benign programs, and 195.00 for ANIMAGUS. In Figure 10c, the average number of unique files written/deleted/renamed per second is between 0.89 to 23.52 for benign programs, and 4.68 for ANIMAGUS. Figure 12 demonstrates the percentage of accessed high-value files out of all accessed files for each program. We can see that 42.44% of files that ANIMAGUS accesses are high-value. Overall, it is difficult to distinguish ANIMAGUS from benign programs by only looking at file-type based metrics.

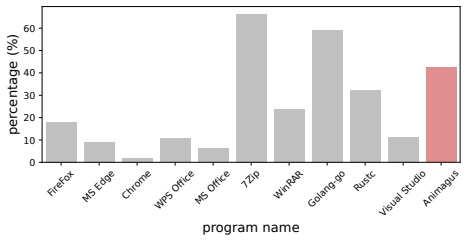


Figure 12: The percentage of accessed high-value files out of all accessed files throughout program lifetime.

Scanning based detector. We investigated three file-type based metrics, i.e., the average number of scanning

operations over program lifetime, the peak frequency of scanning operations over program lifetime, and the average number of unique folders scanned over program lifetime. We analyzed each metric on collected runtime records. Figure 11 presents the overall statistics. In Figure 11a, the average number of scanning operations per second is between 12.00 to 324.33 for benign programs, and 9.98 for ANIMAGUS. In Figure 11b, the average peak frequency of scanning operations is between 157.91 to 9165.00 for benign programs, and 681.50 for ANIMAGUS. In Figure 11c, the average number of unique folders scanned per second is between 0.25 to 12.82 for benign programs, and 3.08 for ANIMAGUS. In addition to static scanning metrics, we also investigate if ANIMAGUS can be detected by a temporal scanning feature. Figure 13 in presents scanning frequency in the first ten seconds when running Visual Studio, 7Zip, MS Edge, and ANIMAGUS. We can see that many benign programs would also scan many folders in their initial phase. Overall, it is difficult to distinguish ANIMAGUS from benign programs by only looking at scanning based metrics.

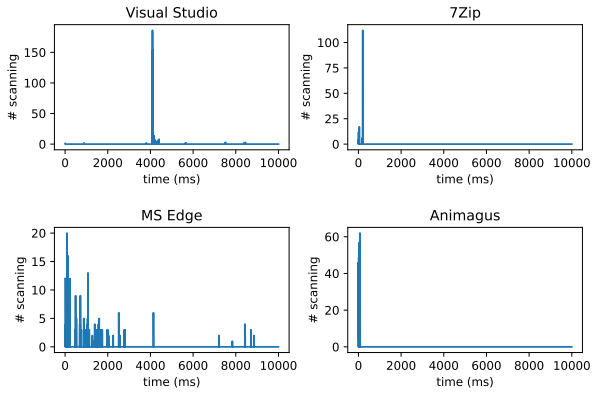


Figure 13: The number of scanning operations in the first ten seconds of different programs' lifetime.

Dummy-access based detector. We investigate the percentage of files accessed in exploitable patterns among all accessed non-high-value files over program lifetime. We analyzed this metric on the collected runtime records and the statistics present in Figure 14. From the figure we can see, the average percentage is between 3.92% to 28.14% for benign programs, and 25.23% for ANIMAGUS. This indicates that it is difficult to distinguish ANIMAGUS from benign programs by only looking at Dummy-access patterns.

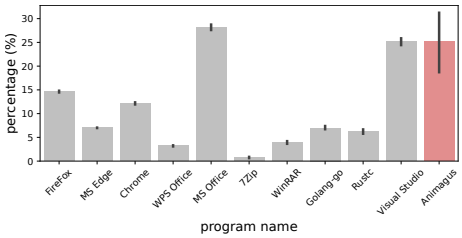


Figure 14: The percentage of files accessed in exploitable patterns among all accessed non-high-value files throughout program lifetime.

B.3. Visualization of Behavior Differences

We also investigate whether the differences make ANIMAGUS more like ransomware or not. We apply Principal Component Analysis (PCA) to quantify the differences. We choose the features of ShieldFS as the metrics. PCA is used to transform the features into a two-dimensional space for visualization. Figure 15 presents the visualization result. As we can see, the behavior distance between ANIMAGUS and its imitated benign program is much shorter than the one between ANIMAGUS and ransomware samples. Therefore, detection tools cannot make a correct decision on the basis of the features.

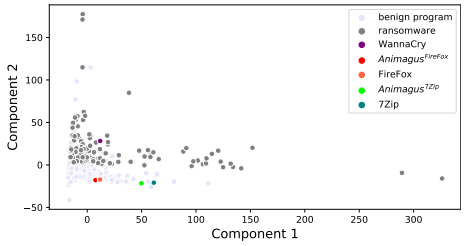


Figure 15: Visualization of behavior features of different programs. The features used here are in line with ShieldFS. The dimensionality reduction is performed by PCA.

B.4. Similarity of Ransomware Families

In section 5.3, we randomly choose one sample from each family to conduct experiments. This is because samples of a family share an almost identical behavioral pattern at runtime. To prove it, we randomly chose four samples from family Avoslocker and one sample each from families Wannacry and Phobos. We evaluated the similarity of their runtime I/O sequences using the Ratcliff-Obershelp Algorithm. Figure 16 presents their similarity. The similarity between four samples from Avoslocker ranges from 0.927 to 0.996. By contrast, the similarity between samples from different families ranges from 0.484 to 0.757. This indicates that samples from a family behave identically at runtime.

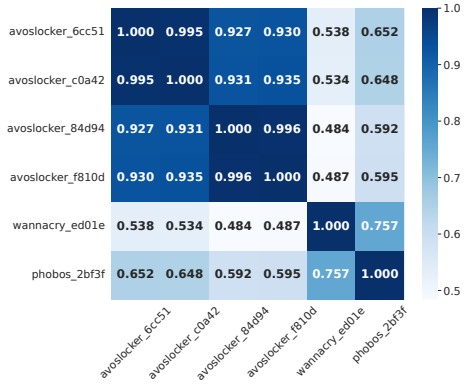


Figure 16: Runtime behavior similarity of ransomware samples from different families.