Detecting Evasive Multiprocess Ransomware

by

Ryan R. LaPointe

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2021

APPROVED:

Professor Lorenzo De Carli, Thesis Advisor

Professor Craig A. Shue, Reader

Professor Craig E. Wills, Head of Department

Abstract

Recent work described techniques that could be used by ransomware to evade behavioral ransomware detectors by using multiple benign-looking processes to cooperatively encrypt files. We designed and evaluated two classifiers that each detect the presence of ransomware that uses those techniques with greater than 99 percent recall and with 100 percent precision. One of the classifiers can also determine which processes are part of the ransomware with greater than 95 percent recall but with a significant trade-off between precision and speed, achieving 92.4 percent precision after hundreds of files are encrypted. We prepared for a user study to collect a new dataset, developing the necessary client and server software.

Acknowledgements

I would like to thank my advisor, Professor Lorenzo De Carli, for providing the idea for this thesis and for his invaluable guidance. I would like to thank the authors of The Naked Sun [1] [2] for sharing necessary datasets and code and for providing advice. I would like to thank Professor Craig A. Shue for being the reader for this thesis. I would like to thank the WPI Computer Science Department for providing computing resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Naked Sun [1] [2] describes techniques that could be used by future ransomware to evade state-of-the-art behavioral ransomware detectors. The authors' prototype ransomware, named Cerberus, successfully evades two state-of-the-art academic detectors, ShieldFS [3] and RWGuard [4], and evaded a commercial anti-ransomware product [5]. With ransomware attacks a booming industry, it is conceivable that ransomware may soon use evasion techniques like those described in The Naked Sun.

The contributions of this thesis are as follows: We replicated some of the results of The Naked Sun and of ShieldFS. We analyzed the ShieldFS dataset and identified a shortcoming in the malicious dataset. We prepared for a user study to collect a new dataset, developing the necessary client and server software. We developed two classifiers that can detect the presence of ransomware that uses the evasion techniques described in The Naked Sun with high precision and recall. The second classifier can also determine which processes are part of the ransomware in the face of the functional splitting technique, but not of the process splitting technique. Determining which processes are part of the ransomware requires many

files to be encrypted in order to have good precision, creating a trade-off between detection speed and precision.

We provide background in §2. Our evaluation of the ShieldFS dataset is in §2.5.1. The replication of the results of ShieldFS is described in §3.4 and the results are in §4.1. The replication of the results of The Naked Sun is described in §3.5 and the results are in §4.2. The first classifier, which we call the *straw man classifier*, is described in §3.3 and evaluated in §4.3. The second classifier, which we call the *new classifier*, is described in §3.6 and evaluated in §4.4. The preparation for the user study is described in §3.7 and §4.5. We discuss the results in §5 and conclude in §6.

Chapter 2

Background

2.1 Behavioral Ransomware Detection

Ransomware is a kind of malware that takes computers or data hostage, denying users access to their computers or data until a ransom is paid to the attacker. Crypto-ransomware is a type of ransomware that denies access to data by encrypting the data such that decryption is possible only by obtaining a decryption key from the attacker after paying the ransom. Other kinds of ransomware exist, including screen lockers, which prevent the user from logging into the infected device until the ransom is paid [6] [7]. Crypto-ransomware has become the predominant form of ransomware in recent years [8]. In this paper, we are concerned only with crypto-ransomware and we hereafter refer to crypto-ransomware simply as 'ransomware'.

Ransomware is a growing problem for businesses. In 2020, a survey [9] conducted by Sophos found that 51 percent of surveyed organizations experienced a ransomware attack in the preceding year. The U.S. Cybersecurity and Infrastructure Security Agency said that ransomware is "costing billions of dollars" per year [10].

As the ransomware 'industry' matures, not only are more organizations being

attacked, but also infrastructure is being built with which to make the ransomware attacks of tomorrow more advanced and scalable. Sophos [9] states the current trend in ransomware attacks is toward "highly-targeted, sophisticated attacks that take more effort to deploy." In addition to more sophisticated attacks, 2020 also brought a rise in ransomware-as-a-service (RaaS) services that lowered the barrier to entry for new cybercriminals to get into the ransomware business [11]. Today, there are even state actors developing ransomware and engaging in ransomware attacks [12].

The traditional approach to malware detection involves static analysis of executable files and pattern-matching against the memory space of processes [13]. These methods are becoming increasingly ineffective in the face of obfuscation, polymorphism, and metamorphism [14] [15], which allow malware to achieve the same malignant outcomes while appearing different to the detector each time it is seen.

Recent work in the detection of ransomware, including ShieldFS [3] and RWGuard [4] which are discussed in §2.4.1, has focused on the use of behavioral characteristics of ransomware processes, such as filesystem activity and use of encryption routines, to detect the presence of ransomware on a machine. Behavioral detection techniques are more difficult for ransomware to evade because the behavior hunted by the detector is semantically related to the goals of the ransomware; for example, for ransomware to encrypt files it must necessarily read those files, encrypt the contents, and write out the encrypted version of the files. For this reason, behavioral ransomware detection is resistant to the obfuscation, polymorphism, and metamorphism techniques that thwart traditional detection techniques and exhibits good recall when faced with previously unseen ransomware.

A downside of behavioral ransomware detection is the increased potential for false positives. The encryption of files is a behavior that is shared by ransomware and many benign programs, such as full-disk encryption programs. In some cases, ran-

somware has been found to install and use a legitimate full-disk encryption program as its encryption mechanism [16], blurring the line between benign and malicious behavior.

Behavioral ransomware detection may, but need not, use machine learning techniques, including supervised learning and unsupervised learning, and may use different kinds of behavioral measurements, such as filesystem activity, system calls, hardware performance counters, decoy files, and more. For reasons that will become apparent in the next section, this report focuses on behavioral ransomware detection through supervised learning on filesystem activity.

2.2 The Naked Sun

The Naked Sun [1] [2] describes techniques that could be used by future ransomware to evade state-of-the-art behavioral ransomware detectors. The authors' prototype ransomware, named Cerberus, successfully evades two state-of-the-art academic detectors [3] [4] and evaded a commercial anti-ransomware product [5]. The authors of The Naked Sun take advantage of an assumption that is implicit to the design of state-of-the-art academic behavioral ransomware detectors—that ransomware can be identified by the behavior of individual processes. Cerberus violates this assumption by using multiple benign-looking processes to cooperatively encrypt the user's files. The authors of The Naked Sun found that dividing the ransomware workload among many processes in such a manner that each process mimics the behavioral characteristics of a benign program allows ransomware to successfully evade the detectors.

The Naked Sun describes three different evasion techniques: *process splitting*, whereby different processes are assigned different groups of files to encrypt; *func-*

tional splitting, whereby different functions, such as directory listing, reading, encrypting, writing, and renaming, are performed by different processes; and *mimicry*, which combines both process and functional splitting such that the ransomware processes emulate the behavioral characteristics of benign processes.

ShieldFS [3], one of the academic detectors that the Cerberus prototype evades, relies heavily on supervised machine learning models trained on the filesystem activity of processes. RWGuard [4], the other academic detector evaded by the Cerberus prototype, uses supervised machine learning on filesystem activity for one component, but also has several other components that use different behavioral features and classification techniques. The component that uses supervised machine learning on filesystem activity is treated as a necessary, but not sufficient, indicator for classifying processes as ransomware. The authors of The Naked Sun chose ShieldFS and RWGuard to test Cerberus against because they "were published in highly visible venues, and in both cases the authors kindly provided enough material (code and/or datasets) and support to enable us to run their software" [1].

Insofar as they evade ShieldFS, the techniques used by the Cerberus prototype are instances of *adversarial example* attacks against supervised machine learning models. The models exhibit *feature vulnerability*; the features do not adequately capture the difference between benign programs and ransomware, leaving the door open for ransomware that appears to be benign as measured by those features. Benefitting from the knowledge that the filesystem activity of individual processes is being fed into a classifier, and of the specific features being used, the Cerberus prototype generates processes that imitate the behavior of benign processes with regard to the features being used by the classifier while still cooperatively encrypting all of the user's files. Indeed, the design of the Cerberus prototype was heavily informed by the design of ShieldFS and the processes generated by Cerberus mimic behavioral

characteristics extracted from the benign training dataset used by ShieldFS.

However, the techniques described in The Naked Sun cannot be written-off as merely whitebox adversarial example attacks against a known model with a known training set. The RWGuard detector has many components besides the single component that uses a model and features similar to ShieldFS. Malwarebytes Anti-Ransomware [5], the commercial anti-ransomware product evaded by the Cerberus prototype, was a black box to the authors of The Naked Sun; they had no knowledge of how it works other than the information posted on Malwarebytes' website.

Therefore, the core contribution of The Naked Sun is not a whitebox attack on a supervised machine learning model. The Naked Sun identifies and exploits an invalid assumption made in the selection of features for the state-of-the-art ransomware detectors. State-of-the-art detectors assume that the behavior of individual processes can be classified as benign or malicious in a vacuum, without needing any other information. Both ShieldFS and RWGuard contain components that look at activity on the machine without distinguishing between processes, but those components are used only as secondary indicators and are not empowered to sound the alarm on their own. In order for ShieldFS or RWGuard to sound the alarm, the detector must be able to recognize a specific process as ransomware. The Naked Sun figuratively asks the question, 'What happens if no individual process appears to be ransomware?' State-of-the-art ransomware detectors fail when that is the case.

2.3 Windows Filesystem I/O Logging

In Microsoft Windows, a request by a user-space process to perform filesystem I/O results in the generation of an I/O Request Packet (IRP) or a Fast I/O Request

(FIO request). A filter driver is a software component that is run in kernel mode and that intercepts IRPs and/or FIO requests as they are processed by the operating system. The filter driver can then transform or analyze the requests. A minifilter driver serves the same purpose as a filter driver, but it interfaces with the Filter Manager, which itself is a filter driver. [17] [18] [19] [20]

The evaluations performed by ShieldFS and The Naked Sun involved logging filesystem activity on Microsoft Windows systems and deriving features for a supervised learning model from those logs. To capture filesystem activity and produce the logs, the ShieldFS authors developed a tool that they named *IRPLogger*. The ShieldFS authors describe IRPLogger as "a low-level I/O filesystem sniffer" of which the core is "a minifilter driver that intercepts the I/O requests generated for each filesystem primitive invoked by userland code (e.g., `CreateFile`, `WriteFile`, `ReadFile`). IRPLogger enriches the raw IRPs with data including timestamp, writes entropy, and PID" [3]. The authors of The Naked Sun reimplemented [21] the IRPLogger tool to use for evaluating their Cerberus prototype.

2.4 Ransomware Detection Techniques

2.4.1 Behavioral Ransomware Detectors

ShieldFS

ShieldFS [3] is one of the academic ransomware detectors that The Naked Sun evades. ShieldFS uses 242 random forest classifiers, each with 100 trees. The features for the classifiers are derived from the IRP traces in the ShieldFS dataset, which is discussed in §2.5. The features are number of folders listed, number of files read, number of files written, number of files renamed, number of files of a given

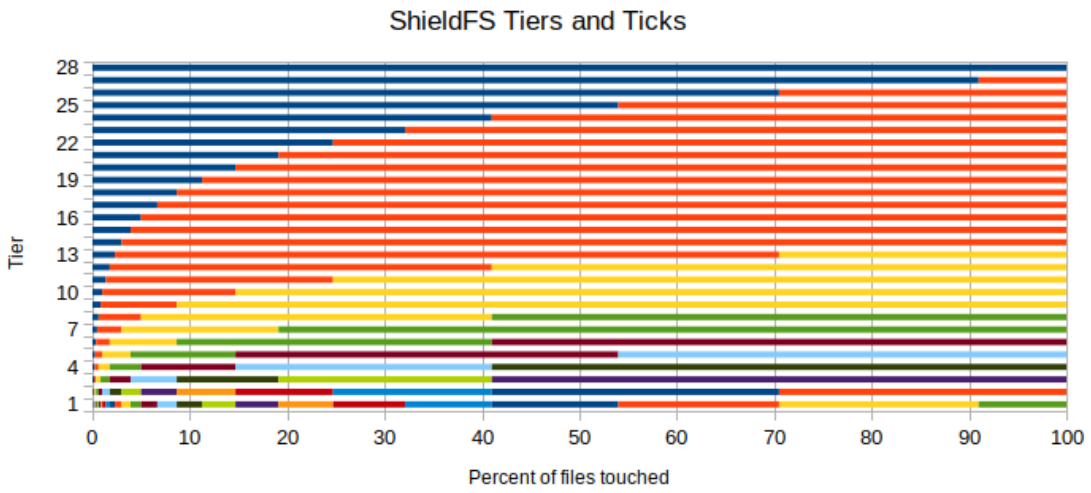

Figure 2.1: The tiers and ticks into which the random forest classifiers used by ShieldFS are arranged. Each bar represents a tier and each differently-colored portion of a bar represents a tick.

file extension accessed, and write entropy. The features are calculated with respect to each process—referred to as *process-centric* features—and among all processes—referred to as *system-centric* features. 121 of the random forest classifiers use the process-centric features and the other 121 random forests use the system-centric features. Within each group of 121 random forests, each random forest uses features calculated over a different window. Windows are defined not with respect to time but with respect to the number of files that have been touched by the relevant process or by all processes. The windows are organized into *tiers* and *ticks* as shown in Figure 2.1.

ShieldFS' primary mechanism for identifying ransomware is the process-centric random forests. According to the ShieldFS authors, "the system-centric model is used only in combination to the process-centric model." "Proccesses can enter a 'suspicious' state when the process-centric classifier is not able to cast a decision. In this case, ShieldFS queries the system-centric model." "The rationale is that the

system-centric model has a good recall for multi-process malware, but has potentially more false positives." [3] The Cerberus prototype developed by the The Naked Sun authors is always classified as benign by the process-centric model, and therefore the system-centric model is not consulted.

ShieldFS also scans for the presence of cryptographic primitives in a process's memory as an additional factor used in its determination.

RWGuard

RWGuard [4] is the other academic ransomware detector evaded by The Naked Sun. Like ShieldFS, RWGuard uses a random forest classifier with features derived from filesystem activity; however, RWGuard also uses several other detection methods.

The *DMon* module "deploys decoy files" that "should not be modified in normal situations" [4]. A process that tries to write to a decoy file is labeled as malicious.

The *PMon* module uses a random forest classifier with features derived from filesystem activity. IRPs and FIO requests are logged using a method virtually identical to that used by ShieldFS and The Naked Sun; the logging is performed on Microsoft Windows by a filesystem minifilter driver that the RWGuard authors call IRPLogger. The features used are the numbers of write, read, open, create, and close requests and the number of temporary files created. The features are per-process.

The *FCMon* module looks for changes in the type, size, entropy, and similarity of files during write operations. Like the *DMon* and *PMon* modules, this module performs classification with respect to processes.

The *FCls* module classifies the activity on a file as benign or malicious without respect to processes. This module is only engaged "after the *PMon* and *FCMon*

modules' detection that a process is making significant changes in the file(s)" [4].

The *CFHk* module analyzes a process' use of `CryptoAPI` library functions.

UNVEIL

UNVEIL [22] is a dynamic analysis system for detecting ransomware. In order to label a program as ransomware or benign, it executes the program in a sandbox environment and monitors filesystem activity and any changes to the machine's graphical user interface. The method used by UNVEIL to monitor filesystem activity is the same as that used by ShieldFS, RWGuard, and The Naked Sun; IRPs are logged by a filesystem minifilter driver. Like ShieldFS, UNVEIL augments the traces with the Shannon entropy of the data being read or written. The UNVEIL paper does not specify the exact features derived from the traces nor the architecture of the classifier into which the features are fed; however, it does specify that UNVEIL compares the entropy of read and write requests to and from the same file offset, and that UNVEIL examines "the I/O access sequence for each file in a given run". UNVEIL achieves a recall of 0.963 and a precision of 1.

Redemption

Redemption [23], the successor to UNVEIL, analyzes the filesystem behavior of processes in real time and can undo the filesystem changes made by detected ransomware, like ShieldFS and RWGuard. Redemption assigns a "malice score" to each process and uses a fixed threshold of the malice score for classifying processes as malicious or benign. The malice score is the weighted average of six different scores derived from different behavioral features. The first three features are categorized by the Redemption authors as *content-based features*, whereas the remaining three features are *behavior-based features*. The content-based features are

the entropy difference between a read and a write to the same part of a file, the portion of a file written with respect to its total size, and deletion of a file. The behavior-based features are the number of files written in a directory, the number of different types of files written, and the frequency of write requests. REDEMPTION achieved a sensitivity of 1 and a specificity of 0.995. The REDEMPTION authors found that the specificity was reduced to only 0.941 when using only the content-based features without the behavior-based features. The I/O performance overhead of REDEMPTION is 2.6% "for realistic workloads" and up to 9% for some workloads. REDEMPTION "typically" reports file encryption and secure deletion applications as malicious.

CryptoDrop

CRYPTODROP [24] is another behavioral ransomware detector that can suspend ransomware processes, although it does not undo the changes made by the ransomware. Like SHIELDFS, RWGUARD, UNVEIL, REDEMPTION, and THE NAKED SUN, a Windows driver is used to intercept and analyze filesystem I/O requests. The behavioral features used by CRYPTODROP are file type changes, dissimilarity between old and new file content, higher average entropy of write operations than of read operations for a process, and file deletions. CRYPTODROP maintains a reputation score for each process and classifies a process as ransomware when the score exceeds a threshold. In the CRYPTODROP authors' testing, CRYPTODROP produced no false negatives out of 492 ransomware samples and one false positive out of 30 benign programs.

2.4.2 Potential Approaches for Detecting Multiprocess Ransomware

Decoy Files

Some academic ransomware detectors [25] [26] [27] [28] use *decoy files*, which are files created for the sole purpose of detecting ransomware. No legitimate purpose exists for a process to access a decoy file, so any process that accesses a decoy file is assumed to be malicious. Despite the limitations [29] of decoy files, the use of decoy files is a promising method for detecting ransomware that uses multiprocess evasion techniques.

The The Naked Sun authors state that decoy files are outside the scope of their work, and that "decoys are a promising strategy, but they raise usability concerns" [1]. One component of RWGuard uses decoy files, but that component was not present in the code shared with the The Naked Sun authors and was not included in their testing.

Graph-Based Intrusion Detection

Graph-based intrusion detection, with its ability to identify relations between processes and correlate their activity, is another promising technique for detecting ransomware that uses multiprocess evasion techniques. Graph-based intrusion detection has been applied to the detection of APT (advanced persistent threat) activity in works including StreamSpot [30], HOLMES [31], and Unicorn [32]. Graph-based intrusion detection could be used to identify related processes that should be grouped together when performing behavioral ransomware detection, allowing the processes that are part of a multiprocess ransomware program to be identified as related and then analyzed by existing detectors as if they were one process. This

would allow existing ransomware detectors to be used for detecting multiprocess ransomware with little modification.

2.5 The ShieldFS Dataset

The authors of The Naked Sun obtained the dataset that was used for training and testing the ShieldFS ransomware detector. The ShieldFS dataset is comprised of IRP traces—logs of all IRPs and FIO Requests—and is divided into ransomware and benign portions. The ransomware portion consists of 383 IRP traces, taking up 19 GiB, that were each captured from a virtual machine while ransomware was executing, covering a total of 383 different ransomware samples obtained from VirusTotal. The benign portion consists of 45,102 IRP traces, taking up 24 GiB, that were captured from 11 real user workstations over a period of several weeks. The ShieldFS dataset also contains information about the sizes of files on the machines from which the data were collected. The ransomware traces contain some simulated benign activity in addition to the ransomware activity in an attempt to make them more realistic. For a description of how the traces were collected, see §2.3.

IRPLogger records, among other things, the name of the process originating each I/O request and the path of the file that is the subject of the operation. These two pieces of information are masked in the ShieldFS dataset for user privacy. The masking is performed by replacing each filesystem path component with a salted hash thereof. The only parts of a path that are exempt from masking are the drive letter, the file extension (the portion of the file name that comes after the final '.' if any is present) and certain whitelisted directories such as `C:\Windows\` and `C:\Program Files\`.

The information about file sizes is masked in a similar manner to the IRP traces.

The masking performed on the IRP traces and on the file sizes information prevents discerning the size of a particular file in a trace. The file sizes information is useful only for statistical purposes, such as calculating the average and standard deviation of file sizes for a particular file extension on a particular machine.

2.5.1 Evaluation of the Simulated Benign Activity in the ShieldFS Ransomware Traces

Regarding the simulated benign activity contained in the ransomware IRP traces, the ShieldFS authors state, "We installed common utilities such as Adobe Reader, Microsoft Office, alternative Web browsers, and media players. ... At runtime, our analysis environment emulates basic user activity (e.g., moving the mouse, launching applications)." We evaluated the simulated benign activity in the ransomware traces by comparing it to the activity in the benign traces.

We measured the average operations per second of each benign program over each trace. For the benign traces, we included all programs located outside of the `C:\Windows\` directory. For the ransomware traces, we listed every program that appears anywhere in any of the 383 traces and we counted the number of traces that each program appeared in. We excluded programs that appeared in only one trace. We excluded programs located under the `C:\Windows\` directory. We then considered the remaining programs one by one and kept only the ones that appear to be benign based on the path of the executable.

Figure 2.2 and Figure 2.3 display the average operations per second of each benign program over each trace for the benign and for the ransomware traces. In our opinion, the simulated benign activity in the ransomware traces leaves much to be desired. Although both the benign and the ransomware charts show a high peak at the far left, the drop after this initial peak is one order of magnitude in the

benign chart whereas it is 2-3 orders of magnitude in the ransomware chart. This means that the portion of the benign processes in the ransomware traces that exhibit very low operations per second is much larger than the corresponding portion in the benign traces. After the drop from the initial peak, both charts are fairly flat until around 1000 operations per second, where both charts drop off.

The operations per second measurements are obtained by taking the total number of operations for a given program in a given trace and dividing by the duration of the trace in seconds. Therefore, there are two possibilities—either the benign processes in the ransomware traces perform less operations per second than the processes in the benign traces or, more likely, the benign processes in the ransomware traces were performing I/O for only a portion of the trace (such as if they were launched and then left idle for the rest of the trace) whereas processes in the benign traces were being actively used by a user for the entire trace.

These potential deficiencies are relevant only for particular uses of the ShieldFS dataset. With regard to ShieldFS, the potential problems with the dataset are relevant only with respect to the training and testing of the system-centric model and not the process-centric model. The authors of The Naked Sun used only the process-centric model when they tested their ShieldFS reimplementation. In this work, we use the system-centric model of the ShieldFS reimplementation as the *straw man classifier* discussed in §3.3. Given the potential issues with the ShieldFS dataset, it is possible that the straw man classifier is over-fitting, as we state in §4.3. Should that be the case, it provides more justification for the new classifier we develop and for the user study we work towards. We believe that the potential issues with the ShieldFS dataset are not relevant to our new classifier.

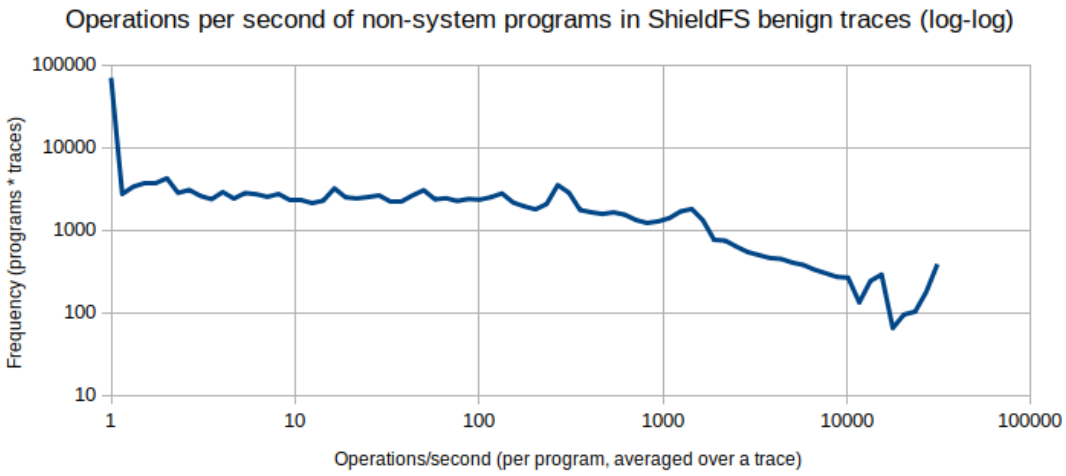

Figure 2.2: Log-log histogram of operations per second of non-system programs in the ShieldFS benign dataset.

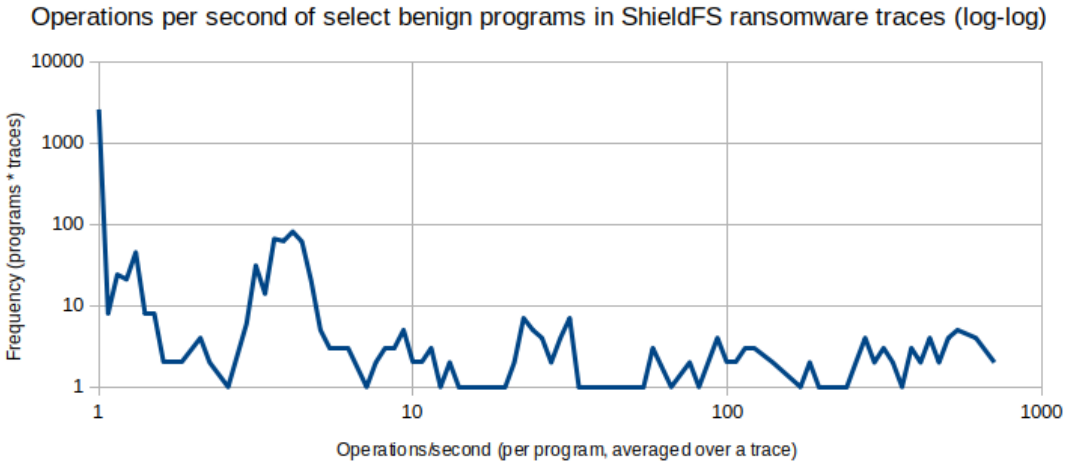

Figure 2.3: Log-log histogram of operations per second of select benign programs in the ShieldFS ransomware dataset.

Chapter 3

Methods

3.1 Definition of the Problem

The goal of this M.S. thesis was to detect the (as-yet hypothetical) category of ransomware that uses the multiprocess evasion strategies described in The Naked Sun to make per-process activity look benign, and that uses an undetectable means for communicating between the cooperating processes. The research question was *To what extent is it possible to detect multiprocess ransomware when each individual ransomware process appears benign and the inter-process communication is undetectable?*

There are a variety of different strategies for covert inter-process communication described in the literature and the methods of detecting them are equally diverse. If we assumed a specific method of inter-process communication, then our work could be defeated by swapping out the inter-process communication technique, and our work would be more about detecting that inter-process communication technique than it would be about detecting ransomware. Having chosen not to assume a specific method of inter-process communication, our remaining alternatives were to

detect all possible means of inter-process communication or to assume that the inter-process communication is undetectable. The former is not straightforward, although graph-based intrusion detection techniques like those used by StreamSpot [30], HOLMES [31], and Unicorn [32] show promise. To keep the task of a reasonable scale for an M.S. thesis, we chose the latter option, assuming undetectable inter-process communication.

More specifically, the goals of this thesis, in order of priority, were:

1. Correctly detect the presence of multiprocess ransomware on the machine.

2. Detect the presence of multiprocess ransomware on the machine quickly enough that the encryption process can be interrupted and the user's data can be saved.

3. Determine which processes are part of the ransomware.

4. Determine which processes are part of the ransomware quickly enough that the encryption process can be interrupted and the user's data saved without causing a denial of service condition for the machine. That is to say, crashing the machine would not be an acceptable way to stop the ransomware under this goal, unlike Goal 2.

It is not necessary to achieve all four of these goals for success. Achieving only the first two would be sufficient.

3.1.1 Threat Model

The threat model for this thesis makes the following assumptions:

- Ransomware may be installed on the machine. If present, the ransomware will try to encrypt the user's files.

- The ransomware operates with only user privileges and cannot gain administrative privileges.
- The ransomware splits its actions between multiple processes in a way such that each process's activity viewed alone does not look like ransomware.
- The processes have access to an undetectable means of inter-process communication.

3.2 Code Obtained from Other Researchers

The authors of The Naked Sun state, "As we could not obtain the original code or a prototype for ShieldFS due to patenting issues, we re-implemented the ShieldFS classifier exactly as described, interacting with the ShieldFS's authors to clarify any potential misunderstanding" [1]. We obtained the ShieldFS reimplementation from the The Naked Sun authors and adapted parts of it for this work, as described in §3.3.

The authors of The Naked Sun also reimplemented [21] the IRPLogger tool described in ShieldFS. We used the IRPLogger reimplementation as the starting point for developing the user study client software, which we describe in §3.7.

We obtained an IRP trace of the Cerberus prototype from the authors of The Naked Sun.

We obtained the trained EnCoD [33] model in HDF5 format from the EnCoD authors, and we integrated the model into the user study client software.

3.3 Straw Man Classifier

As a first, low-hanging-fruit approach to solving the problem, we decided to try running the system-centric half of ShieldFS alone, independent of the process-centric half. We refer to this as the *straw man classifier* in this paper. Because the ShieldFS authors had been unwilling to share their code with the authors of The Naked Sun for patent-related reasons, the authors of The Naked Sun had reimplemented ShieldFS. We obtained this reimplementation of ShieldFS and used it as the starting point to build this solution. After our modifications, the ShieldFS reimplementation is 1915 lines of Python code.

The results of testing the straw man classifier against the ShieldFS dataset and against the Cerberus prototype are in §4.3.

3.3.1 Development Process

The code we received from the authors of The Naked Sun consisted of several Python scripts. Each script was designed to be run iteratively or in parallel, passing different parameters each time to tell the script which small part of the ShieldFS dataset to process that time. After modifying the scripts to work with our copy of the ShieldFS dataset, we added progress bars to the scripts and made them process the entire dataset (benign or ransomware) in one go. We then tried running the scripts. Table 3.1 shows the amounts of time that it took for us to run the scripts. These times were achieved by running the scripts on a virtual machine with 8 vCPUs and 4GB of memory. We aborted the system-centric scripts because they were showing estimated time remaining of multiple weeks. It was clear that we needed to rework the system-centric feature calculation procedures to have higher performance, maybe by multithreading.

Script	Duration
benign process-centric	1 day, 18:05:14
ransomware process-centric	1 day, 8:20:32
benign system-centric	Aborted after 4 days, 3:15:41
ransomware system-centric	Aborted after 2 days, 4:15:36

Table 3.1: Run times for the ShieldFS reimplementation.

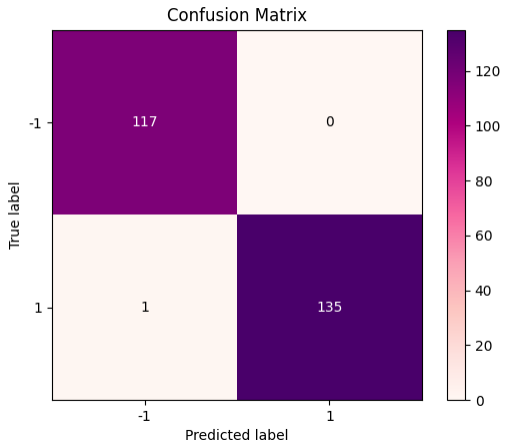

Figure 3.1: The confusion matrix for tier 1, tick 6 of the process-centric model of ShieldFS.

After the process-centric scripts were done running, we tried running the process-centric half of the ransomware detector. The model training and testing procedure output a series of confusion matrices as `PNG` images, one for each of the 121 random forests. One of the confusion matrices is displayed in Figure 3.1. The ransomware detector built, trained, and tested all 121 of the individual random forest classifiers that make up the process-centric half of the ShieldFS detector, but it did not perform the final step of the ShieldFS detector, which is using the outputs of those models to render a single overall verdict of whether a process is ransomware or benign. In fact, we realized that the feature calculation scripts (the ones that took a long time to run) had not preserved information necessary to correlate between the 121 different classifiers using the method described in ShieldFS.

We asked the authors of The Naked Sun about the apparent incompleteness

of the ShieldFS reimplementation. They explained that they had taken the arithmetic average of the accuracy of the 121 process-centric random forests to arrive at their accuracy metric.

We modified the feature calculation procedures to add the machine ID and process ID to each feature vector so that the results of the 121 classifiers could be correlated. We then modified the ransomware detector to remove the machine IDs and PIDs from the feature vectors when training the models and to use them to correlate the results of the classifiers within each tier. Per the ShieldFS paper, if K consecutive ticks within a tier label a process as ransomware, then ShieldFS labels the process as ransomware. Otherwise, ShieldFS labels the process as benign. The ransomware detector now outputs a confusion matrix for each tier and a single overall ShieldFS confusion matrix in addition to the confusion matrices for each (tier, tick) tuple that it already made.

We performed a large amount of refactoring on the codebase for usability reasons.

We tried to improve the performance of the feature calculation through multithreading. After modifying the benign process-centric feature calculation procedure to use multiple threads, we tested it and was surprised to find that despite using 7 threads to calculate the features, the process overall was only using 120% CPU as reported by `top`. Before, when the feature calculation was done with a single thread, the reported CPU usage was 100%, so we had expected using 7 threads to result in 700% CPU usage unless there was a disk bottleneck. We checked the disk activity and the disk was clearly not causing a bottleneck. We did some performance profiling of the code with `cProfile` and the results did not help explain what was happening. We then discovered while searching online that multithreading in Python does not work how we expected it to: in Python, multiple threads cannot execute at the same time. Python itself is not thread-safe. Multiprocess-

ing is the only way for a Python program to truly execute on multiple CPU cores simultaneously.

We modified the feature calculation procedures to use multiprocessing instead of multithreading. We decided to use the initial process as a supervisor process that spawns worker processes to do the feature calculation. Each worker process performs the feature calculation for a trace and then tells the supervisor process that it is ready for another trace. The supervisor then gives the worker another trace to work on. If the supervisor has no traces left to hand out, it tells the worker to die. Once all workers exit, the overall procedure is complete.

The multiprocessing approach was successful, with each worker process using 100% CPU as reported by `top`. We configured the program to use one less worker process than there are CPU cores on the machine. The supervisor process is asleep most of the time and consistently uses less than 1% CPU as reported by `top`. This leaves most of one CPU core's capacity available so that the system does not lag or hang.

After completing the conversion to multiprocessing, we ran the feature calculation again. This time we ran it on a physical machine with 64 CPU cores and 503GB of memory. Therefore, 63 worker processes were used. Table 3.2 shows the amounts of time that it took for us to run the feature calculation. We do not know why the ransomware system-centric feature calculation takes so much longer than the ransomware process-centric feature calculation. We deemed the feature calculation fast enough for our needs.

Script	Duration
benign process-centric	02:10:36
ransomware process-centric	00:19:52
benign system-centric	02:12:58
ransomware system-centric	16:44:38

Table 3.2: Run times for the ShieldFS reimplementation.

3.4 Replication of Results of ShieldFS

Before testing the straw man classifier against the techniques described in The Naked Sun, we performed some tests to confirm that the ShieldFS reimplementation remained faithful to ShieldFS after our modifications. The results of the tests are in §4.1.

For most of our tests, the benign and ransomware feature vectors were combined into one pool that was randomly split into 70% training / 30% test. For the tests where "1-machine-off cross-validation" is indicated, the following procedure was used: For the benign dataset, one machine's feature vectors were used as the test set and the rest were the training set. For the ransomware dataset, the feature vectors of 42 randomly-chosen traces were the test set and the rest were the training set. This was repeated 11 times so that each of the 11 benign machines was used for the test set once.

We tested the process-centric and the system-centric halves of ShieldFS separately because ShieldFS [3] does not provide enough detail about how the two halves are integrated in order for us to reimplement the integration.

3.5 Replication of Results of The Naked Sun

Before testing the straw man classifier (the system-centric half of the ShieldFS reimplementation) against the techniques described in The Naked Sun, we per-

formed some tests to see if we could reproduce the results of The Naked Sun with regard to the process-centric half of the ShieldFS reimplementation.

The authors of The Naked Sun were able to evade the process-centric classifier in multiple ways, which they categorize into *process splitting*, *functional splitting*, and *mimicry*. Within the functional splitting category, the methods were: splitting the activity of the ransomware into four groups—directory list (DL), read (RD), write (WT), and rename (RN)—and further dividing each group into 5 separate processes; splitting the ransomware activity into only two groups of (DL,RD) and (WT,RN) and further dividing each group into 10 processes; and splitting the ransomware activity into two groups of (DL,RN) and (RD,WT) and further dividing each group into 10 processes. Each of these three methods resulted in complete evasion of the process-centric model (recall of 0). The mimicry category was comprised of the Cerberus prototype. [1]

We tested the process-centric classifier in six test cases: first, we trained and tested the process-centric classifier with the unmodified ShieldFS dataset as a control; second, we trained the process-centric classifier with the unmodified ShieldFS dataset and then tested it with (DL,RD), (WT,RN) splitting; third, we trained the process-centric classifier with the unmodified ShieldFS dataset and then tested it with (DL,RN), (RD,WT) splitting, fourth, we trained and tested the process-centric classifier with (DL,RD), (WT,RN) splitting; fifth, we trained and tested the process-centric classifier with (DL,RN), (RD,WT) splitting; and sixth, we trained the process-centric classifier with the unmodified ShieldFS dataset and then tested it against Cerberus (mimicry). We varied the number of processes per group. For the tests where the same dataset was to be used for training and testing, we used a 70%/30% train/test split. $K = 3$ was used in all cases.

The results of the tests are in §4.2.

3.6 New Classifier

The straw man classifier detects whether ransomware is present on a machine but it does not provide any insight as to which processes comprise the ransomware. After testing the straw man classifier, we designed a new classifier with the goal of not only detecting the presence of ransomware that uses the techniques described in The Naked Sun, but also identifying which processes comprise the ransomware.

In order to avoid having the same weakness as the ransomware detectors that The Naked Sun evades, we must not calculate features on a per-process basis. Instead, we calculate features on a per-file basis. We classify files as attacked or not attacked and then correlate the results of the per-file classifier to label processes as malicious or benign.

Computing features for files instead of processes tightens the semantic relationship between the features used by the detector and the objectives of the ransomware. As explained in §2.1, the advatange of behavioral ransomware detection over more traditional methods of ransomware detection is the semantic relationship between the goals of the ransomware and the behavior hunted by the detector. For ransomware to encrypt files it must necessarily read those files, encrypt the contents, and write out the encrypted version of the files. By focusing our detector on what is happening to files, we make it more difficult for ransomware to avoid detection. In particular, the functional splitting, process splitting, and mimicry techniques described in The Naked Sun will not fool the file classifier.

The file classifier is a random forest of 100 trees that uses features calculated over a 10-second window from an IRP trace.

3.6.1 File-Based Features

The features used by the file classifier are byte-wise percent of file read during the window, byte-wise percent of file written during the window, byte-wise percent of file read and then written in that order, file entropy at the end of the window, and file entropy delta over the window. The features were chosen to have a high semantic relation to the goal of ransomware to encrypt files. To achieve its goal, ransomware must read the entire file and then write out the encrypted version of the file. It must also either delete or overwrite the unencrypted version of the file.

Byte-wise Percent of File Read During the Window

This value is the number of byte positions read divided by the length of the file.

Byte-wise Percent of File Written During the Window

This value is the number of byte positions written divided by the length of the file. If the file is deleted during the window, that pegs this value to 100%.

Redemption [23] uses the portion of a file written with respect to its total size as one of the features for their detector, and they find that their detector has an I/O performance overhead of 2.6% for realistic workloads.

Byte-wise Percent of File Read and Then Written, in That Order

This value is the number of byte positions that were read and then written, in that order, divided by the length of the file. If the file is deleted during the window, that counts as a write of all byte positions.

File Entropy at the End of the Window

This value is calculated by starting at the end of the window and working backwards, greedily collecting write operations until for every byte position in the file, you have the latest write operation in the window that operated on that byte position, then calculating a byte-wise weighted average entropy for the file.

File Entropy Delta Over the Window

This value is calculated as follows: First calculate the file entropy at the beginning of the window by starting at the beginning of the window and greedily collecting read operations until for every byte position in the file, you have the earliest read operation in the window that operated on that byte position, then calculating a byte-wise weighted average entropy for the file. Then calculate the delta by subtracting the beginning entropy from the end entropy.

Redemption [23] compares the entropy of reads and writes to the same byte positions in a file as one of the features for their detector, and they find that their detector has an I/O performance overhead of 2.6% for realistic workloads.

3.6.2 Dataset Limitations

The original intention had been to use the dataset gathered by the user study described in §3.7; however, for reasons explained in §3.7 the user study was not yet conducted. Therefore, we used the ShieldFS dataset for testing the new classifier. The ShieldFS dataset does not contain the lengths of files, so we assumed that the highest byte position of the file observed being accessed during the window is the last byte of the file. The expected effect of this assumption is to decrease the accuracy of the classifier, so our accuracy measurements are still valid as a lower

bound.

3.6.3 Correlation of File Classifier Output to Classify Processes

We assign a malice score to each process, like REDEMPTION [23]. Each time a file is classified as attacked by the file classifier, we increment the malice score of each process that read the file during the window and then increment the malice score of each process that wrote (or deleted) the file during the window. If a process both read and wrote the file during the window, then its score increases by two. We classify a process as ransomware when its malice score exceeds a threshold.

3.6.4 Implementation

The classifier is implemented in 1097 lines of Python code. The file classifier is implemented using the random forest classifier from the `scikit-learn` Python package. The feature generation is a parallel operation that will use as many processes as there are CPU cores on the machine. During feature generation, each IRP trace is loaded entirely into a SQLite in-memory database, which is then queried to extract the features for that trace. This architecture was used for ease of development. REDEMPTION [23] uses features that are similar to the features that we use, so we take REDEMPTION's performance measurements as evidence that our features can be calculated with similar performance and we do not attempt to achieve such performance in our implementation.

3.7 User Study

The ShieldFS dataset does not contain the sizes of files as noted in §3.6.2. The features we use in our new classifier depend on the sizes of files, so we planned to conduct a new user study to collect a better dataset. Given the potential issues with the ShieldFS dataset identified in §2.5.1, it is possible that the straw man classifier is over-fitting, as we state in §4.3, providing more reason to collect a new dataset.

We developed the necessary software for the user study, including a client application to be installed on subject machines and a web server application that receives collected data from the clients and makes the data available through a REST API.

We were not, unfortunately, able to conduct the user study because during the final correctness testing of the software following the completion of software development, we discovered a problem that halted progress on the user study for over two months. The entropy measurements and EnCoD results appeared to be incorrect. After performing controlled tests with low, medium, and high entropy test files as well as a variety of real files of different types, we determined that the entropy measurements and EnCoD results were always off by one line in the trace. The entropy value and EnCoD results for a given operation were actually for the immediately previous operation. We determined that timing did not affect the issue. We determined that, at runtime, the buffer from which the entropy and EnCoD results were calculated contained the data of the immediately previous operation instead of the data of the current operation.

We replicated the problem using the the IRPLogger tool [21] written by the authors of The Naked Sun, which we had used as a starting point for building the client application. Therefore, the problem was not a result of our code, but was

inherited from the IRPLogger. We informed the authors of The Naked Sun that we believed there was a defect in their code and they investigated the issue. They were unable to isolate the defect and they believe the defect is not in their code.

We still do not know the cause of the problem, but based on both our and The Naked Sun authors' investigations, we believe that the effect of the defect can be corrected after the fact. We therefore implemented a workaround in the server application that applies a correction operation to every IRP trace received from the clients. Unfortunately, by the time the authors of The Naked Sun finished investigating the defect, there was not sufficient time left for this author to proceed with conducting the user study.

3.7.1 Client Software

Requirements

- The software should have an installer that installs the client software without user interaction.
- The software should start running automatically after installation.
- The software should have an uninstaller that uninstalls the client software without user interaction.
- The software should run automatically at boot.
- The software should collect data periodically.
- The software should upload the collected data to the server.
- The software should collect data both when the machine is actively in use and when the machine is idle.

- The software should notify all users of the machine about the user study.

Data to Collect

The user study client software periodically decides to perform a data collection. When a data collection is performed, all filesystem activity on the machine is logged. For each I/O request, the following data are recorded: the type of filesystem operation; a timestamp; the process ID; the process name, the size of data buffer / number of bytes involved in the filesystem operation; the entropy of the data; the file path; and the output of the EnCoD [33] model, which indicates whether the data being read/written appear to be encrypted, compressed, or neither. Before collecting filesystem activity, the client software records the size of each file on the machine. During a data collection, the software records how many users are logged into the machine. The purpose of this is to determine whether the machine is in active use or is idle.

Machine identifiers including network card MAC addresses are recorded. These are used for correlating traces from the same machine in case the client software is reinstalled. The computer name is also recorded, for more conveniently identifying a machine if there is a problem with the client and the client needs to be reinstalled on one particular machine.

Anonymization / Privacy Protection

The names of files within the `C:\Users` directory should not be available to the researchers because those filenames may contain private information of users. Because the machines on which the client software will be installed are WPI-owned machines, the names of files elsewhere on the system will not contain any private information.

For the names of the files that are the subject of I/O operations, for the names of the processes that performed the I/O, and in the lists of file sizes, all paths that begin with `C:\Users\` are masked to protect users' privacy using the following procedure: Each path component (directory name or file name) after the `C:\Users\` is SHA-256 hashed with a salt that is specific to the machine. For a given machine, the same salt is always used after it is initially generated upon the first start of the user study client. The salt is never sent to the server, but a SHA-256 hash of the salt is sent to the server each time a trace is uploaded. The file extension (the part of the filename that comes after the last '`.`' if any is present) is appended (along with a preceding '`.`') to the hash of the filename, so the extension is not protected.

Architecture

The client software is comprised of 4649 lines of C and C++ code.

`UserStudyDriver.sys` is a filesystem minifilter driver that logs I/O Request Packets and Fast I/O Requests to produce the IRP traces.

`UserStudyApp.exe` is the userland application that retrieves the IRP traces from the driver, collects the other data, and uploads collected data to the server. It normally runs as a Windows service named `UserStudyApp` but it can also be invoked from the command line for debugging purposes. It also has a command-line flag that causes it to just display a message box and then exit instead of its usual activity.

`UserStudyInstallerHelper.exe` is a program that assists with the installation and uninstallation process. It is invoked by the installer.

User Study Notice

While the user study client software is installed, every user that logs into the machine is presented with a pop-up message box explaining the user study and stating

that the user should use a different machine if the user does not want to participate in the study. This is accomplished using the registry key `HKLM\Software\Microsoft\Windows\CurrentVersion\Run`. A value is added to that registry key causing `UserStudyApp.exe` to be called with the command-line argument `/w` whenever a user logs in. When `UserStudyApp.exe` is called with the `/w` argument, it uses the `MessageBox` [34] Windows API function to display the user study message box and then exits.

Main Loop

The main loop of the `UserStudyApp` service follows the following procedure, depicted in Figure 3.2: The program sleeps for t seconds and then decides whether to collect data using the decision procedure *should_collect_data* defined below. It then collects data if it decided to do so. Then the program checks if there are any data collections stored on disk that have not yet been uploaded to the server. If any exist, the program attempts to upload all such data collections. Regardless of whether the previous operations succeeded, the program now repeats this procedure, starting by sleeping another t seconds. The value t is configured on the server and retrieved from the server by the client.

If the program is instructed to terminate while sleeping or collecting data, the program exits promptly. If the program is instructed to terminate while uploading data, the program tries to finish uploading data before terminating.

Trace Collection

Each time a data collection is performed, the collection runs until one of the following conditions occurs:

- 100MB (before compression) has been written to the IRP trace file

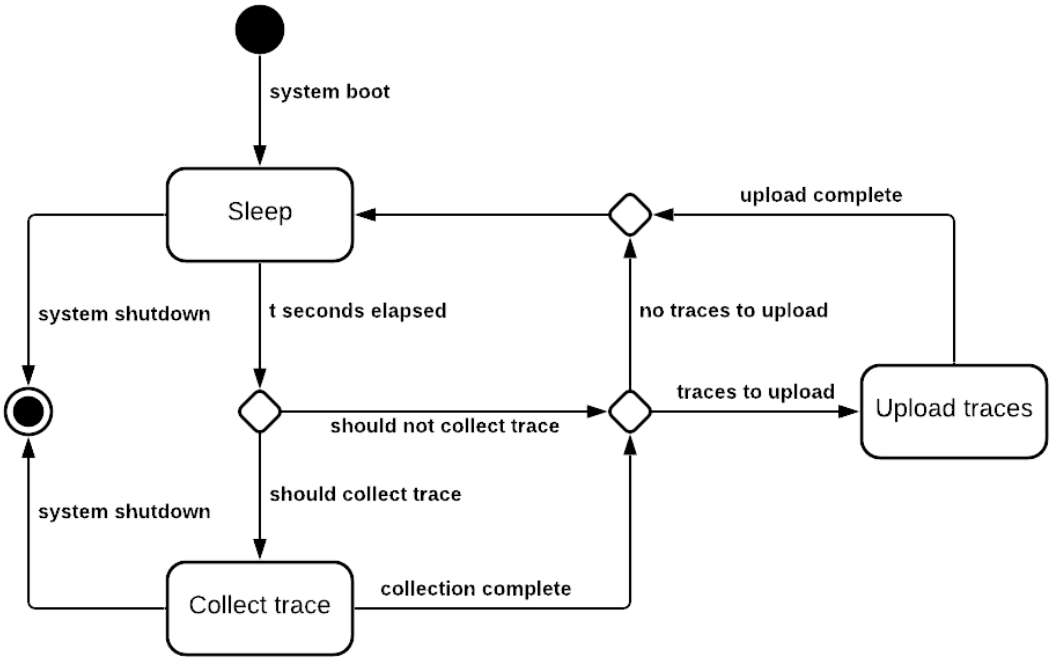

Figure 3.2: Activity diagram of the main loop of the user study client program.

- the collection has been going for 15 minutes
- the collection is interrupted, for example by a system shutdown

At the beginning of data collection and every minute while collecting data, the program retrieves the number of active sessions on the machine. These numbers are recorded. If the median of these measurements for a given data collection is at least 1, then the data collection is considered an *active* data collection for the purposes of this program. If the median is 0, then the data collection is considered an *idle* data collection.

The program keeps count of how many idle data collections and how many active data collections it has collected over a 24-hour sliding window.

Trace Uploading

After collecting data, the program uploads any data collections that have not been uploaded yet. After successfully uploading a data collection, the program deletes it from the disk. Each trace is compressed before being uploaded.

should_collect_data Procedure

The *should_collect_data* decision procedure is as follows: Check the number of active sessions on the machine. If the number of active sessions is greater than 0 and the number of active data collections in the last 24 hours is less than 10, or if the number of active sessions is 0 and the number of idle data collections in the last 24 hours is less than 5, collect data with 0.25 probability. Otherwise, do not collect data.

This procedure limits the rate at which data collections are produced to ten active data collections per machine per day and five idle data collections per machine per day. The sleep time t and the 0.25 probability of collecting data both function to spread out the data collections so they are not all collected during a short period of time each day.

Installer

The installer for the client software is made using the Visual Studio Installer Projects extension [35]. The Visual Studio extension produces both EXE and MSI installers. Both can be run silently (without user interaction) using the `/q` flag.

The installer copies the client software's files to the installation directory, which is `C:\Program Files\Worcester Polytechnic Institute\User Study\` by default. It then invokes `UserStudyInstallerHelper.exe`. `UserStudyInstallerHelper.exe` calls `InstallHinfSectionW` [36] to install the `UserStudyDriver.sys` driver and create the `UserStudyApp` service. `UserStudyInstallerHelper.exe` then starts the

`UserStudyApp` service. The installer also creates a value in the `HKLM\Software\Microsoft\Windows\CurrentVersion\Run` registry key so that the user study warning message box will be displayed upon user login.

The uninstaller first invokes `UserStudyInstallerHelper.exe`, which calls `InstallHinfSectionW` [36] to uninstall the `UserStudyDriver.sys` driver and delete the `UserStudyApp` service. The uninstaller deletes the value in the `HKLM\Software\Microsoft\Windows\CurrentVersion\Run` registry key that was created by the installer. The uninstaller deletes the client software's files.

3.7.2 Server Software

Requirements

- The server should receive and store the collected data from the client.
- The server should have a web interface that provides the researchers with enough information to tell whether everything is working properly / if a client is broken.
- The server should provide an API that the researchers can use to retrieve the collected data from the server.
- The web interface and API should be accessible only to authenticated users.

Architecture

The server runs a web application written in Python using the Django framework. The application is comprised of 1115 lines of Python and HTML code.

Web Interface

The web interface displays information about the machines enrolled in the study and the number of data collections that have been received. The web interface also allows the user to create API keys. The web interface can be accessed using user accounts with username/password.

API for Retrieving Collected Data

The web application provides a REST API that users can use to retrieve data that have been uploaded by the client software. The REST API is accessible only to users with the correct permissions.

Determining Which Machine is Submitting a Data Collection

When the client software submits collected data, it identifies itself with a UUID. This UUID is generated by the client software at the first time it is started following installation on a machine. The client software calls the `/api/client/ping` endpoint on startup and before each time it attempts to submit a data collection. The `/api/client/ping` call contains the UUID and other identifying information about the machine. When the client calls `/api/client/data-collection` to submit a data collection, the UUID it provides must have been previously used in a call to `/api/client/ping`; otherwise, the server rejects the request.

When a client calls `/api/client/ping` using a UUID that the server has not seen before, the server inspects the other identifying information. If any network card MAC addresses are provided and at least one matches a machine that the server has talked to before, then the new UUID is associated with that existing machine. Otherwise, a new machine object is created and associated with the UUID. This procedure ensures that uninstalling and reinstalling the client software does not

cause the server to think that the same machine is multiple different machines.

The computer name is not used for the purpose of matching different UUIDs to the same machine. It is for human use in case it is convenient to identify a machine by computer name. Whenever the computer name changes, the name is updated in the server's database and no record of the previous name is kept.

Storage of Collected Data

IRP traces and file sizes lists are stored as files in the server's filesystem and served by whatever web server program the server is using. All other data are stored in the database (whatever DBMS the web application is using) and retrieved by interacting with the REST API provided by the web application. The file names of IRP traces and file sizes lists contain random strings to make guessing the file names infeasible. For a user to download an IRP trace, the user must obtain the file name from the REST API.

Chapter 4

Results

In §3.1, we state four specific goals for this thesis. In this chapter, we present our results. The straw man classifier and the new classifier both accomplish Goal 1, which is detecting the presence of multiprocess ransomware. The new classifier accomplishes, and the straw man classifier somewhat accomplishes, Goal 2, which is detecting the ransomware quickly enough that the encryption process can be interrupted and the user's data can be saved. The new classifier partially accomplishes Goal 3, which is determining which processes make up the ransomware. The new classifier only slightly accomplishes Goal 4, which is identifying the ransomware processes quickly enough that they can be stopped and the user's data can be saved without taking severe measures like halting the machine.

The research question was *To what extent is it possible to detect multiprocess ransomware when each individual ransomware process appears benign and the interprocess communication is undetectable?* We determine that the presence of such ransomware on a machine can be detected with high recall, precision, and speed, but our classifier has a significant precision/speed trade-off for determining which processes make up the ransomware.

4.1 Replication of Results of ShieldFS

Table 4.1 displays the accuracy of the ShieldFS reimplementation along with the accuracy of the original ShieldFS implementation as stated in ShieldFS [3] and the accuracy of the reimplementation as stated in The Naked Sun [1]. The results in Table 4.1 lead us to believe that the ShieldFS reimplementation, after our modifications, is faithful to ShieldFS. The sensitivity and specificity of the process-centric half of the reimplementation are close to the values given in ShieldFS. §4.1.1 contains the confusion matrices that were used to produce the metrics in Table 4.1.

Tester	Process/system	Train/test method	K	Recall	Precision	Specificity	Accuracy
ShieldFS	combined	10-fold cross-validation	6	0.9974		1	
ShieldFS	combined	10-fold cross-validation	5	0.9974		0.99981	
ShieldFS	combined	10-fold cross-validation	4	0.9974		0.99981	
ShieldFS	combined	10-fold cross-validation	3	1		0.99962	
ShieldFS	combined	10-fold cross-validation	2	1		0.99924	
ShieldFS	combined	10-fold cross-validation	1	1		0.99792	
The Naked Sun	process-centric	1-machine-off cross-validation	N/A				0.986
This work	process-centric	1-machine-off cross-validation	3	1	0.996	0.99999	0.99999
This work	process-centric	70%/30% train/test	6	0.761	1	1	0.998
This work	process-centric	70%/30% train/test	5	0.892	1	1	0.999
This work	process-centric	70%/30% train/test	4	0.961	1	1	0.9997
This work	process-centric	70%/30% train/test	3	1	1	1	1
This work	process-centric	70%/30% train/test	2	1	1	1	1
This work	process-centric	70%/30% train/test	1	1	0.962	0.9997	0.9997
This work	system-centric	70%/30% train/test	6	0.714	1	1	0.996
This work	system-centric	70%/30% train/test	5	0.882	1	1	0.998
This work	system-centric	70%/30% train/test	4	0.969	1	1	0.9995
This work	system-centric	70%/30% train/test	3	0.997	1	1	0.99996
This work	system-centric	70%/30% train/test	2	1	1	1	1
This work	system-centric	70%/30% train/test	1	1	0.882	0.998	0.998

Table 4.1: Accuracy metrics for ShieldFS from three sources: ShieldFS, The Naked Sun, and our tests.

4.1.1 Detailed Results

Process-Centric Confusion Matrices

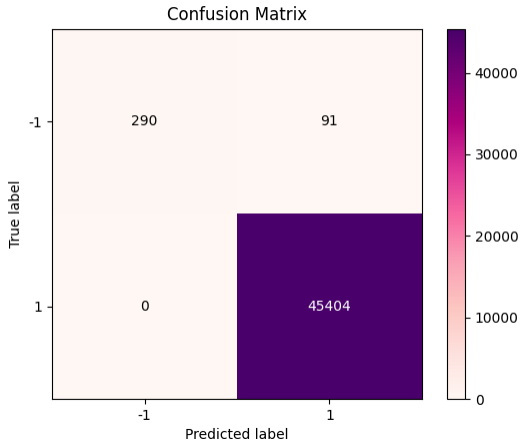

Figure 4.1: Confusion matrix for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 6$ and 70%/30% train/test split.

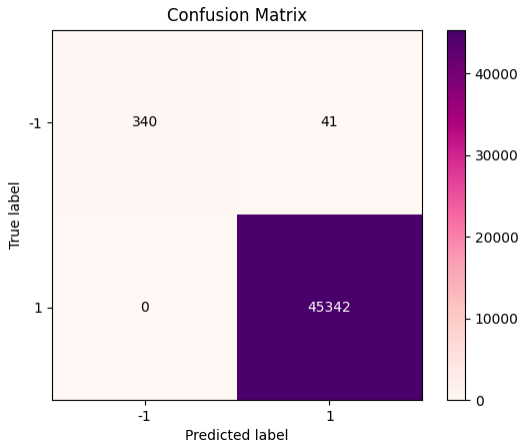

Figure 4.2: Confusion matrix for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 5$ and 70%/30% train/test split.

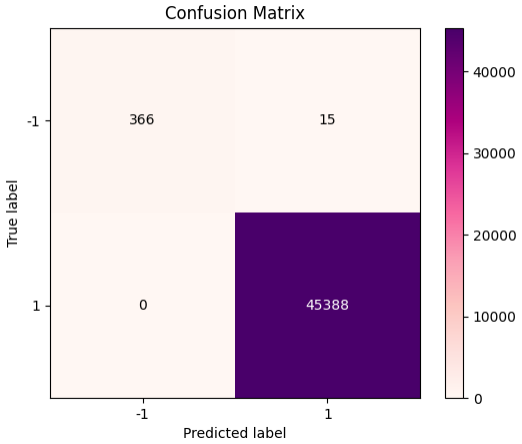

Figure 4.3: Confusion matrix for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 4$ and 70%/30% train/test split.

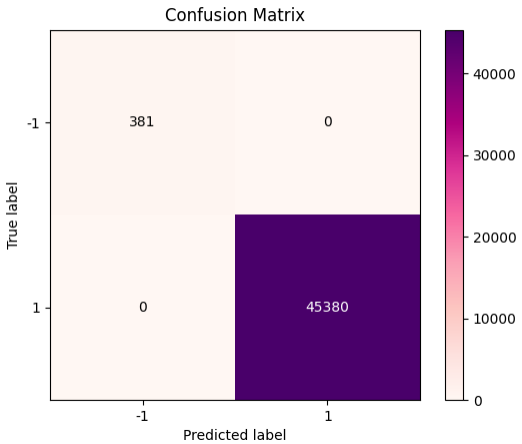

Figure 4.4: Confusion matrix for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

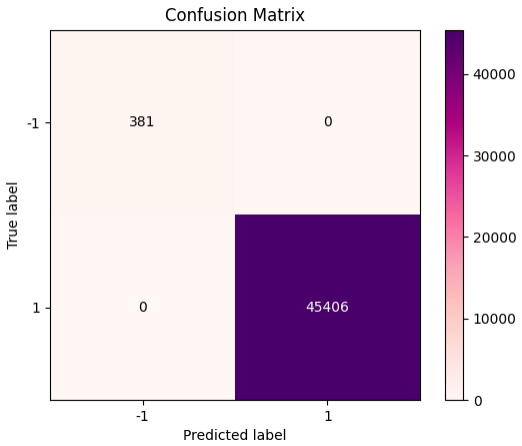

Figure 4.5: Confusion matrix for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 2$ and 70%/30% train/test split.

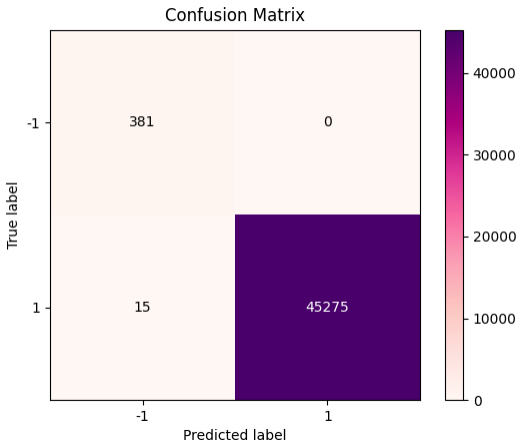

Figure 4.6: Confusion matrix for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 1$ and 70%/30% train/test split.

System-Centric Confusion Matrices

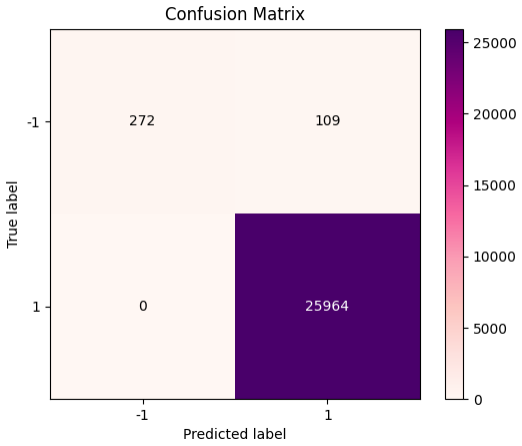

Figure 4.7: Confusion matrix for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 6$ and 70%/30% train/test split.

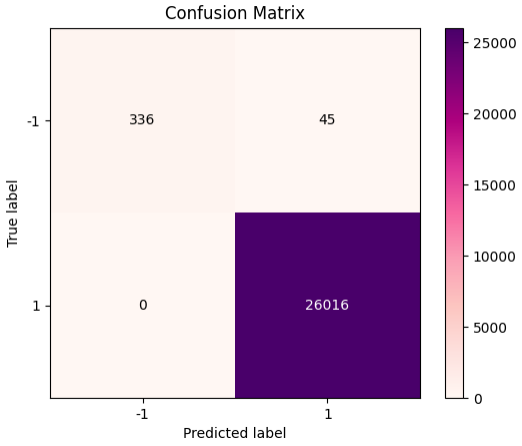

Figure 4.8: Confusion matrix for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 5$ and 70%/30% train/test split.

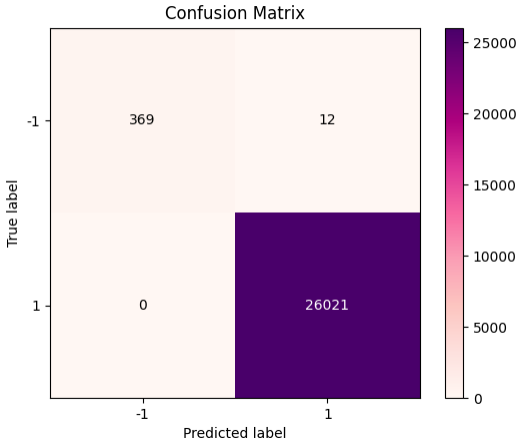

Figure 4.9: Confusion matrix for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 4$ and 70%/30% train/test split.

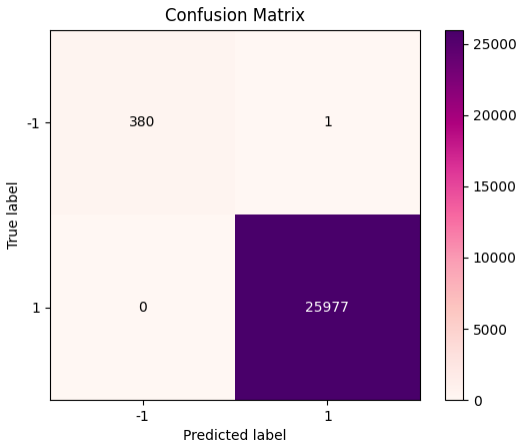

Figure 4.10: Confusion matrix for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

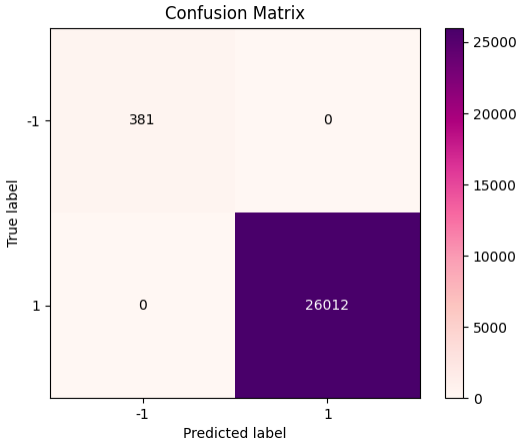

Figure 4.11: Confusion matrix for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 2$ and 70%/30% train/test split.

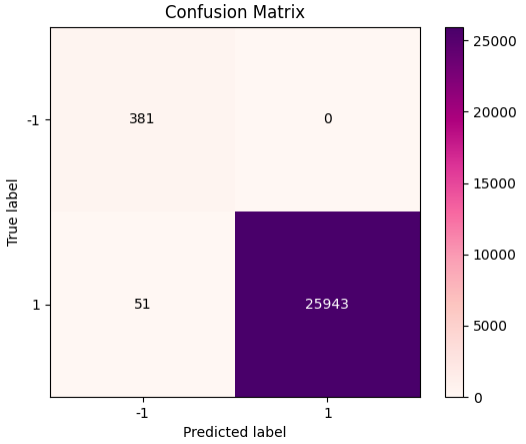

Figure 4.12: Confusion matrix for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 1$ and 70%/30% train/test split.

Splitting	Proc. per group	Recall
None (control)	N/A	1
(DL,RD), (WT,RN)	1	0.469
(DL,RN), (RD,WT)	1	0.430
(DL,RD), (WT,RN)	2	0.307
(DL,RN), (RD,WT)	2	0.281
(DL,RD), (WT,RN)	5	0.283
(DL,RN), (RD,WT)	5	0.318
(DL,RD), (WT,RN)	10	0
(DL,RN), (RD,WT)	10	0

Table 4.2: Results for the process-centric model of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with the functional splitting evasion technique.

4.2 Replication of Results of The Naked Sun

Table 4.2 and Figure 4.13 show the results of training the process-centric half of the ShieldFS reimplementation on the ShieldFS dataset and then testing it against the functional splitting. Table 4.3, Figure 4.14, and Figure 4.15 show the results of training and testing the process-centric half of the ShieldFS reimplementation on the functional splitting. Figure 4.16 shows the results of testing the process-centric half of the ShieldFS reimplementation against the Cerberus prototype. More detailed results, including detection speed, feature importance, confusion matrices, and accuracy for the individual random forests that comprise the ShieldFS detector, are in §4.2.1.

The functional splitting techniques successfully decrease the recall of the process-centric model. The effect is of the same degree as expected. Cerberus successfully evades the process-centric model for all $K \geq 2$ but is detected for $K = 1$.

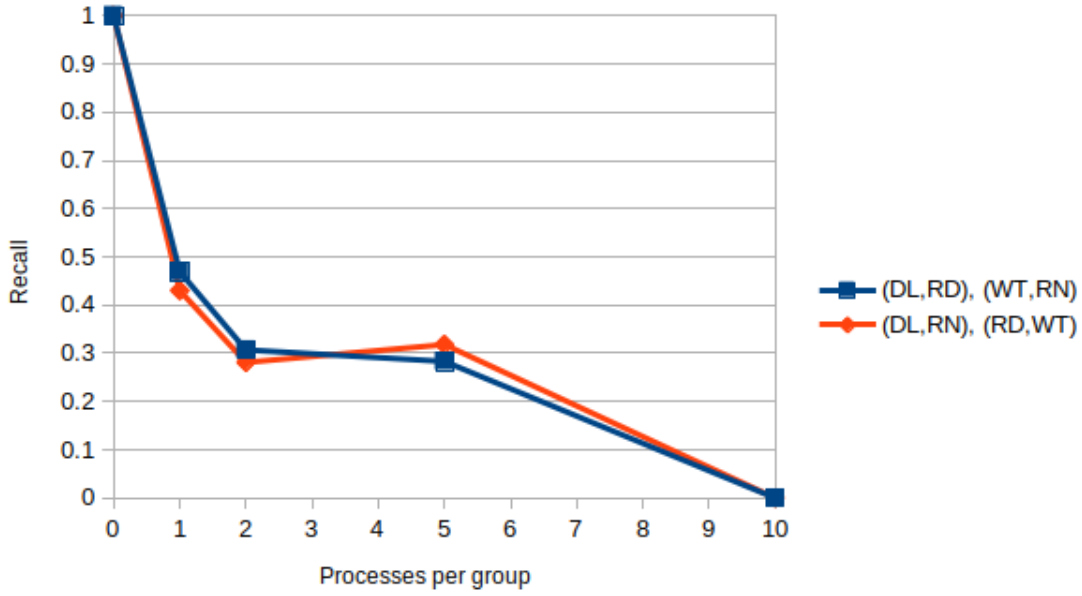

Figure 4.13: Recall for the process-centric model of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with the functional splitting evasion technique.

Splitting	Proc. per group	Recall	Precision	Specificity	Accuracy
None (control)	N/A	0.997	1	1	0.99998
(DL,RD), (WT,RN)	10	0.952	1	1	0.993
(DL,RN), (RD,WT)	10	0.953	1	1	0.993
(DL,RD), (WT,RN)	20	0.922	1	1	0.981
(DL,RN), (RD,WT)	20	0.916	1	1	0.979
(DL,RD), (WT,RN)	40	0.864	1	1	0.947
(DL,RN), (RD,WT)	40	0.844	1	1	0.938
(DL,RD), (WT,RN)	80	0.753	1	1	0.862
(DL,RN), (RD,WT)	80	0.718	1	1	0.840
(DL,RD), (WT,RN)	160	0.572	1	1	0.694
(DL,RN), (RD,WT)	160	0.525	1	1	0.659
(DL,RD), (WT,RN)	320	0.377	1	1	0.485
(DL,RN), (RD,WT)	320	0.353	1	1	0.467

Table 4.3: Results for the process-centric model of the ShieldFS reimplementation when trained and tested with the functional splitting evasion technique.

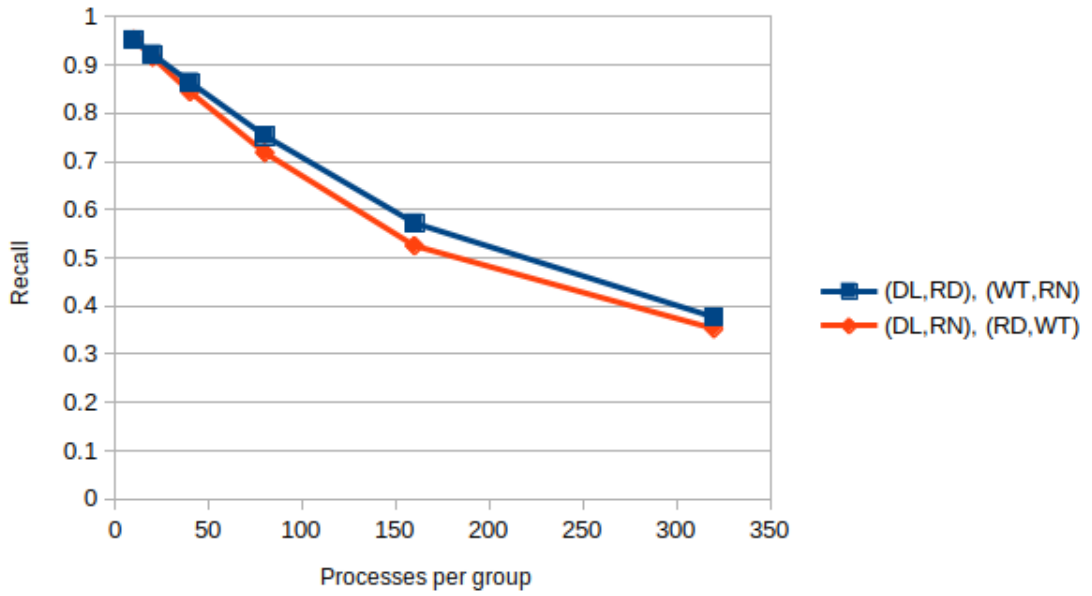

Figure 4.14: Recall for the process-centric model of the ShieldFS reimplementation when trained and tested with the functional splitting evasion technique.

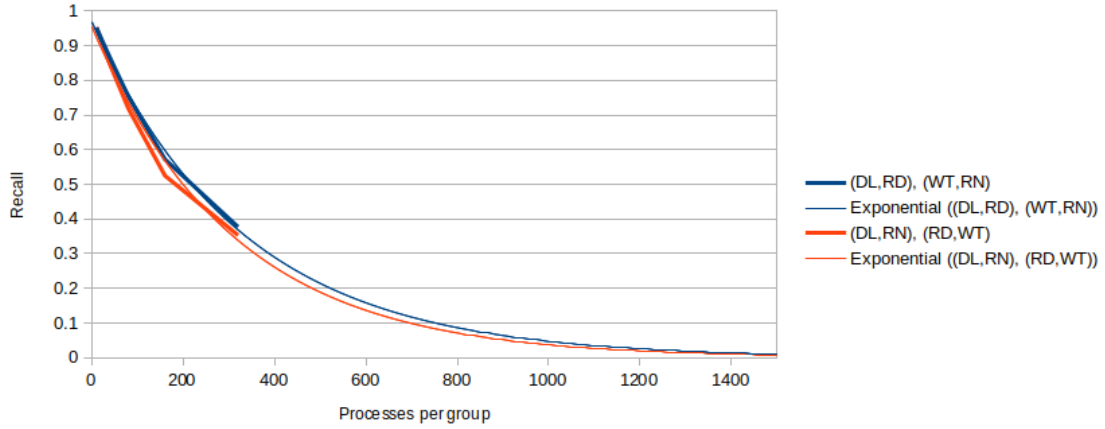

Figure 4.15: Exponential regression of recall for the process-centric model of the ShieldFS reimplementation when trained and tested with the functional splitting evasion technique.

PID	tier	tick	label
2696	1	0	1
2696	1	1	1
2696	1	2	1
2696	1	3	1
2696	1	4	1
2696	1	5	1
2696	1	6	1
2696	1	7	1
2696	1	8	1
2696	1	9	1
2696	1	10	1
2696	1	11	1
2696	1	12	1
2696	1	13	-1
2696	1	14	1
2696	2	0	1
2696	2	1	1
2696	2	2	1
2696	2	3	1
2696	2	4	1
2696	2	5	1
2696	2	6	-1
2696	3	0	1
2696	3	1	1
2696	3	2	1
2696	3	3	1
2696	3	4	-1
2696	4	0	1
2696	4	1	1
2696	4	2	1
2696	5	0	1
2696	5	1	1
2696	5	2	-1
2696	6	0	1
2696	6	1	1
2696	7	0	1
2696	7	1	-1
2696	8	0	1
2696	9	0	1
2696	10	0	1
2696	11	0	1
2696	12	0	1
2696	13	0	-1
2696	14	0	-1
2696	15	0	-1

Figure 4.16: Labels output by the 121 random forests for process-centric half of the ShieldFS reimplementation when tested against Cerberus. If $K \geq 2$, then Cerberus is not detected.

4.2.1 Detailed Results

Training and Testing the Process-Centric Model with the ShieldFS Dataset (Control)

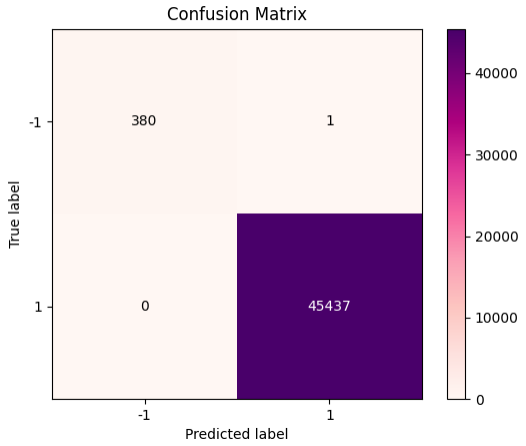

Figure 4.17: Confusion matrix for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

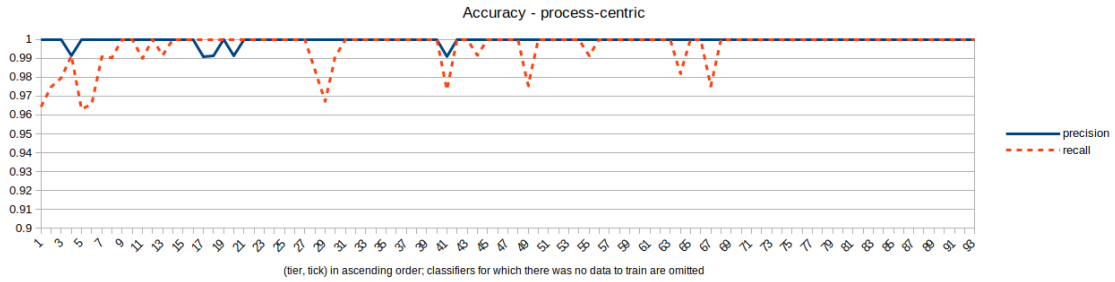

Figure 4.18: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

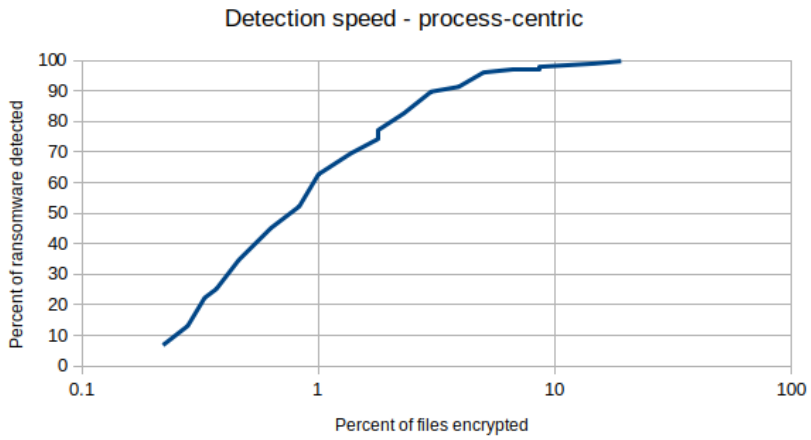

Figure 4.19: Detection speed for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

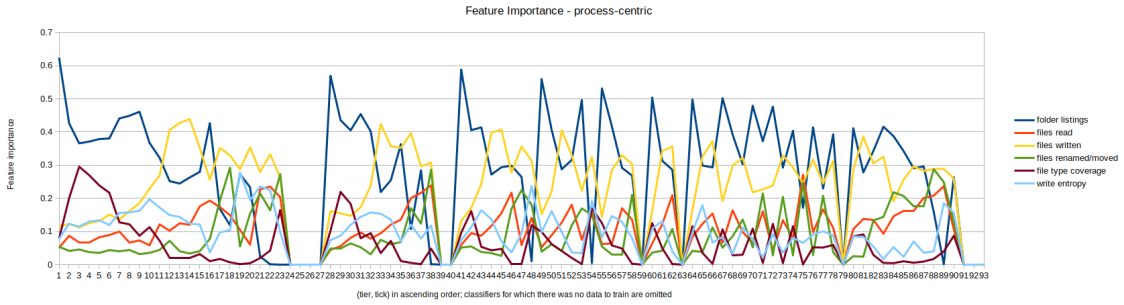

Figure 4.20: Feature importances for the 121 random forests for the process-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

Testing the Process-Centric Model with (DL,RD), (WT,RN) Splitting with 1 Process per Group

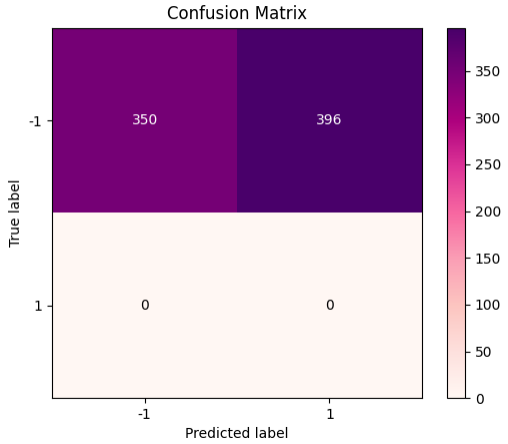

Figure 4.21: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 1 process per group.

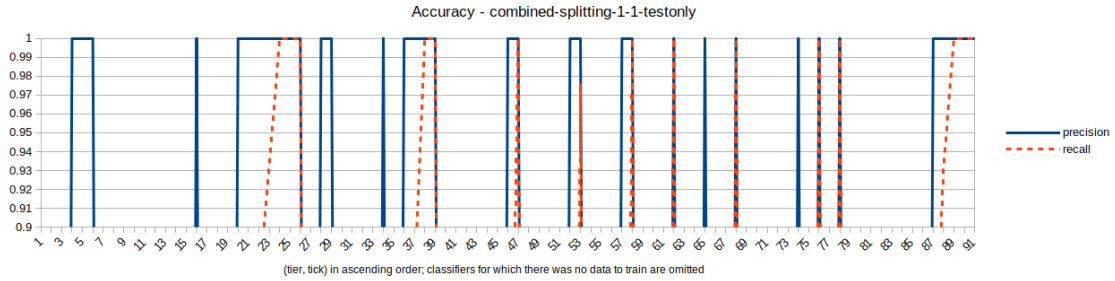

Figure 4.22: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 1 process per group.

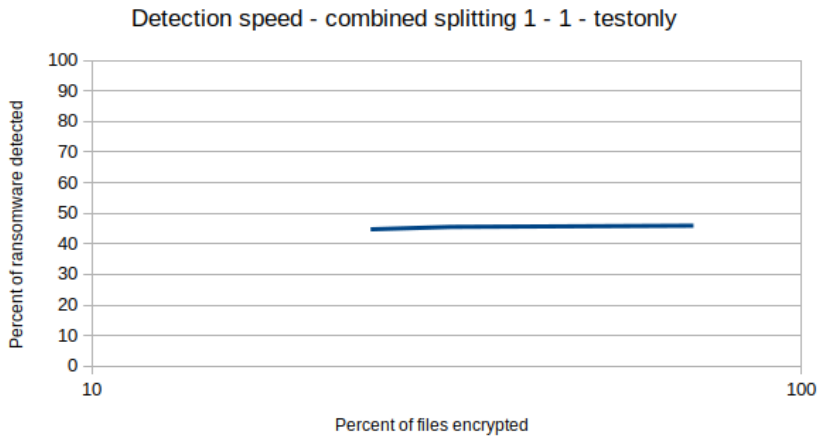

Figure 4.23: Detection speed for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 1 process per group.

Testing the Process-Centric Model with (DL,RN), (RD,WT) Splitting with 1 Process per Group

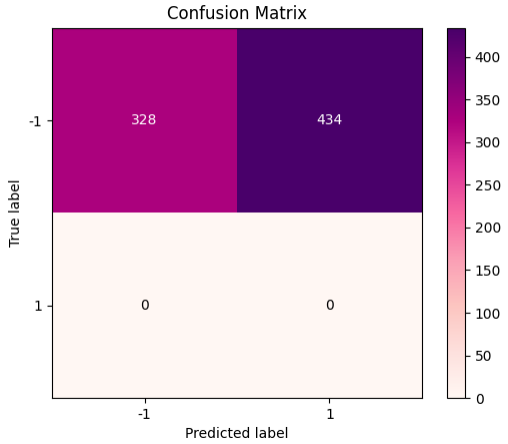

Figure 4.24: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 1 process per group.

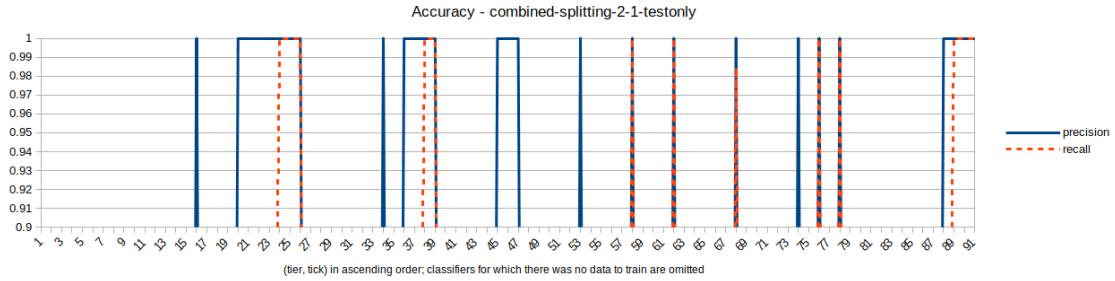

Figure 4.25: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 1 process per group.

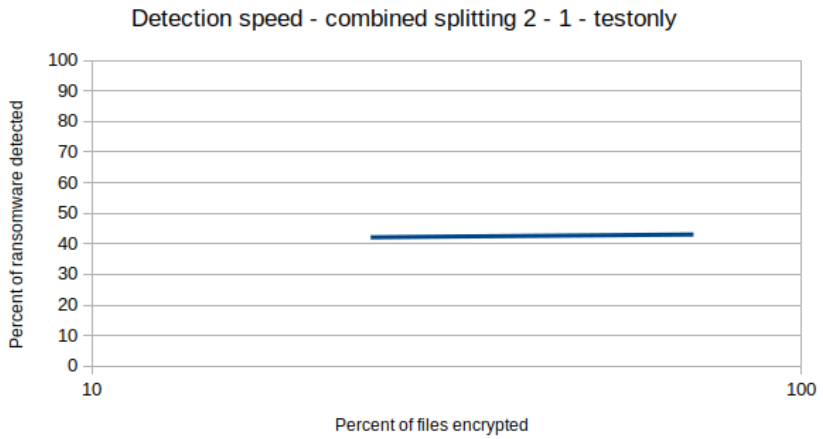

Figure 4.26: Detection speed for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 1 process per group.

Testing the Process-Centric Model with (DL,RD), (WT,RN) Splitting with 10 Processes per Group

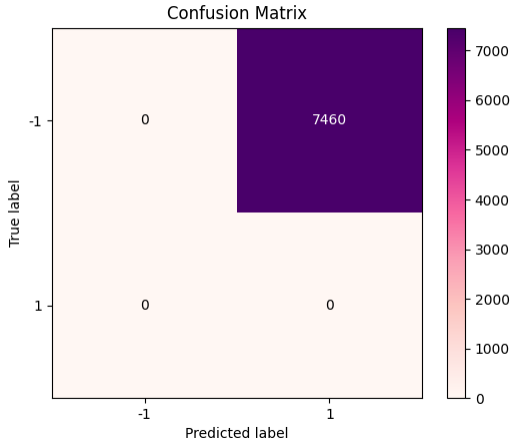

Figure 4.27: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

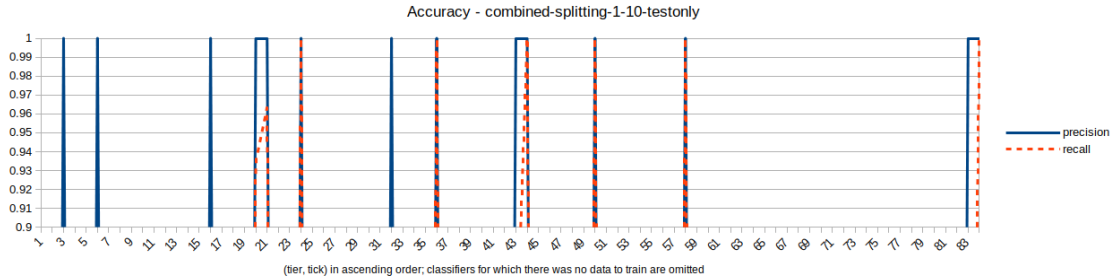

Figure 4.28: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

Testing the Process-Centric Model with (DL,RN), (RD,WT) Splitting with 10 Processes per Group

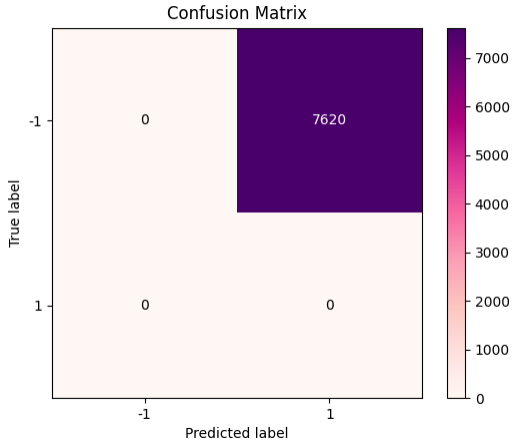

Figure 4.29: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

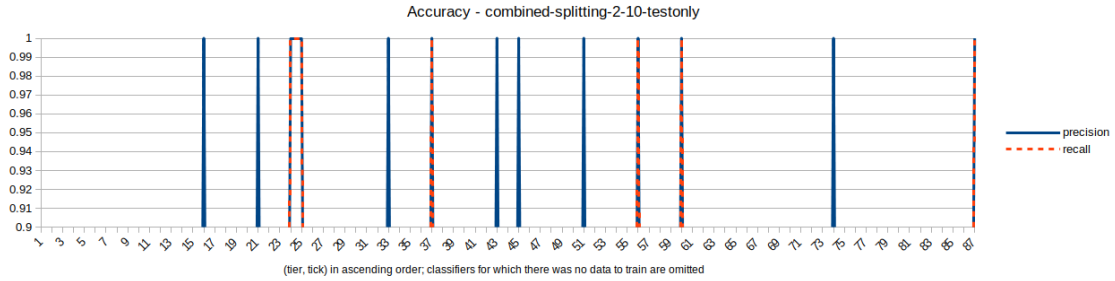

Figure 4.30: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained with the unmodified ShieldFS dataset and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

Training and Testing the Process-Centric Model with (DL,RD), (WT,RN) Splitting with 10 Processes per Group

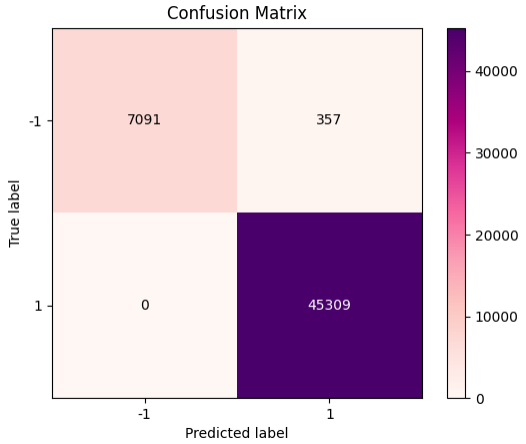

Figure 4.31: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

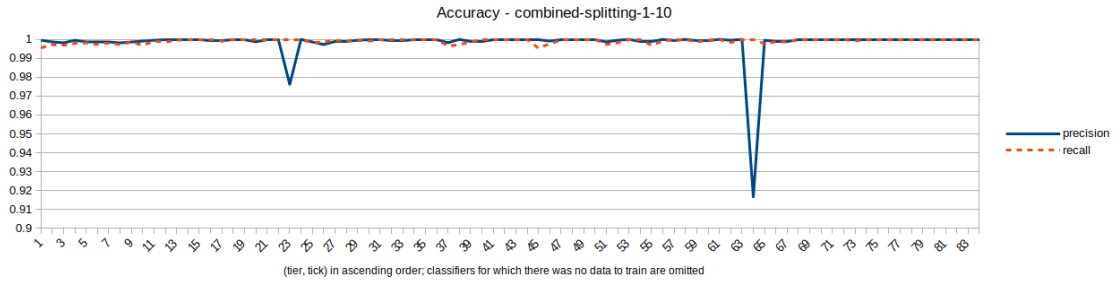

Figure 4.32: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

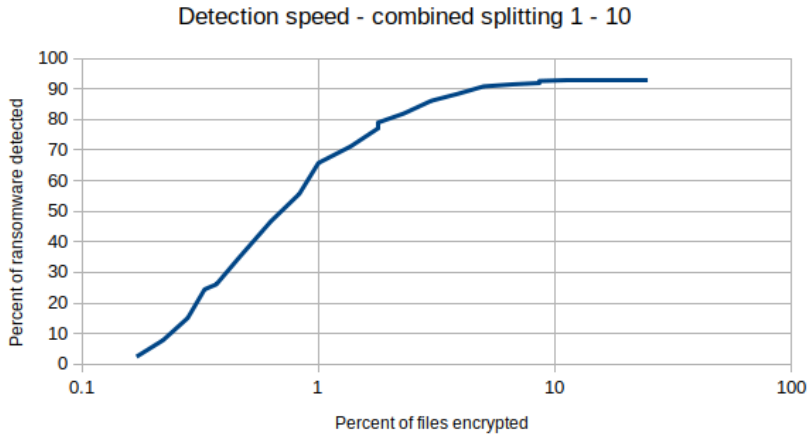

Figure 4.33: Detection speed for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

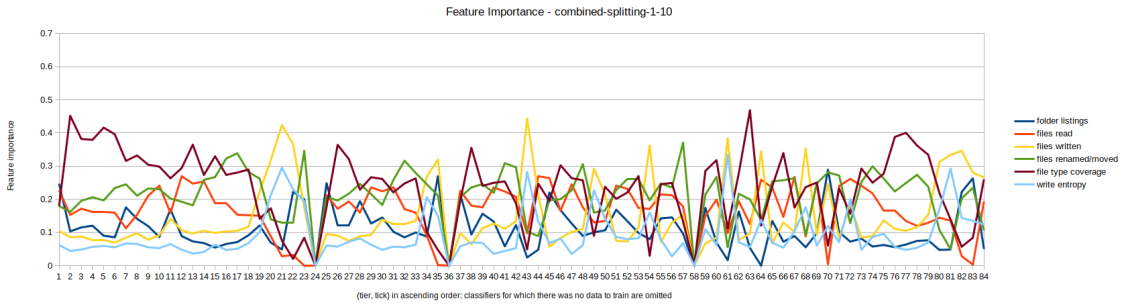

Figure 4.34: Feature importances for the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

Training and Testing the Process-Centric Model with (DL,RN), (RD,WT) Splitting with 10 Processes per Group

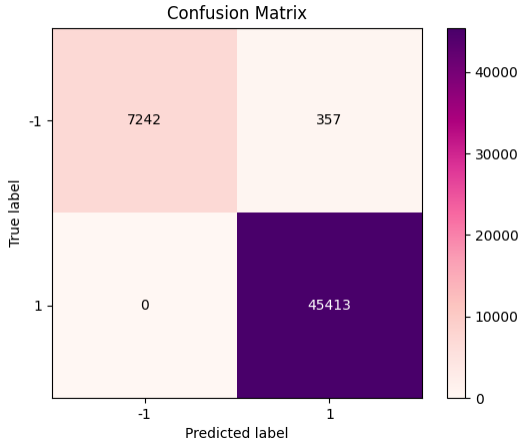

Figure 4.35: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

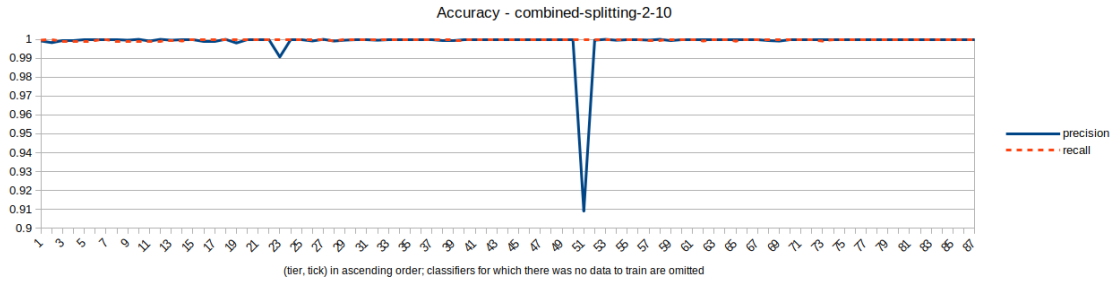

Figure 4.36: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

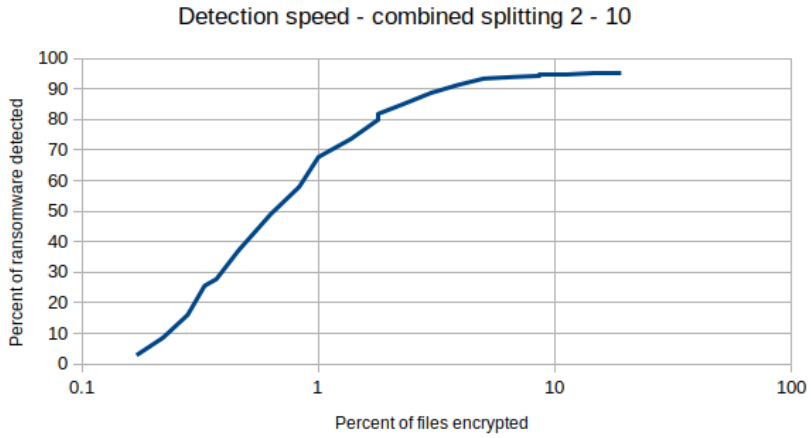

Figure 4.37: Detection speed for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

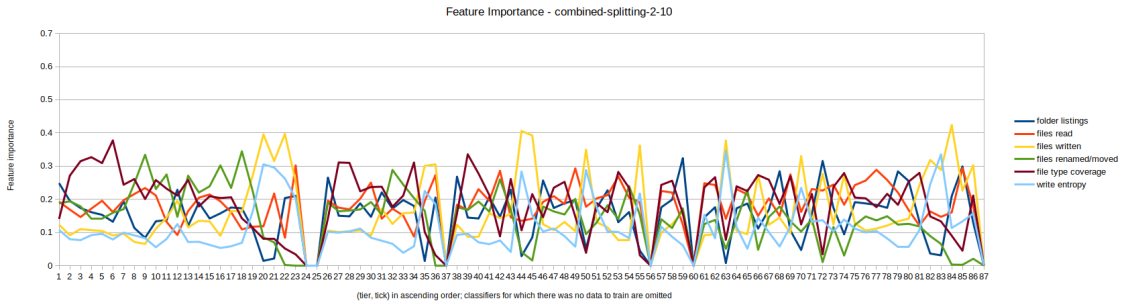

Figure 4.38: Feature importances for the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 10 processes per group.

Training and Testing the Process-Centric Model with (DL,RD), (WT,RN) Splitting with 80 Processes per Group

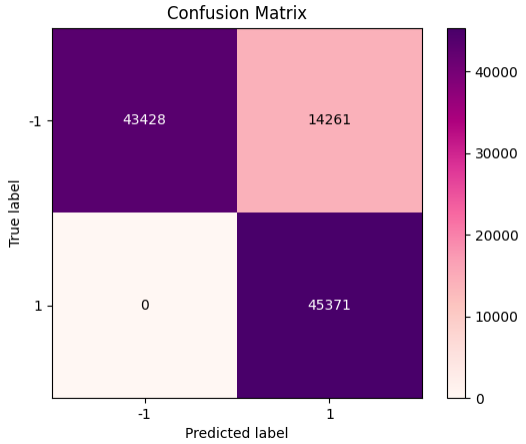

Figure 4.39: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

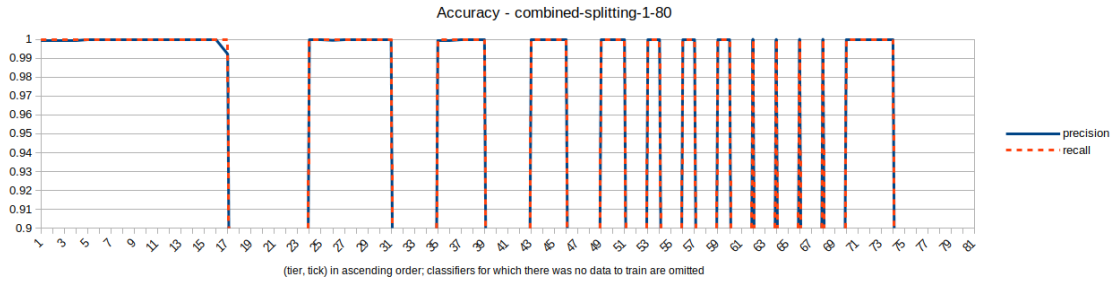

Figure 4.40: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

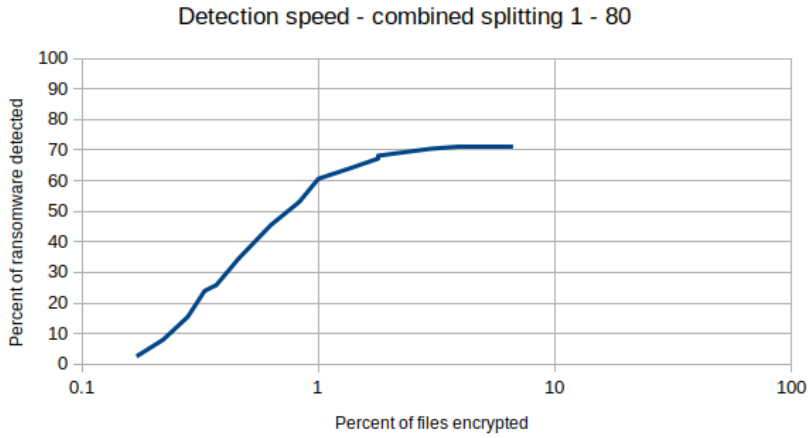

Figure 4.41: Detection speed for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

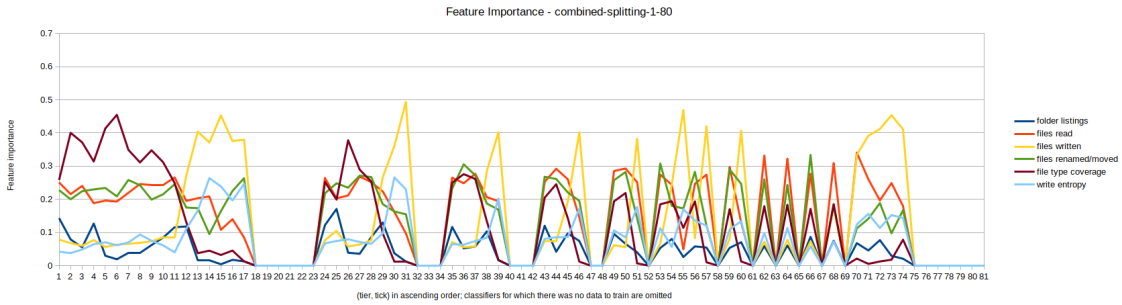

Figure 4.42: Feature importances for the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RD), (WT,RN) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

Training and Testing the Process-Centric Model with (DL,RN), (RD,WT) Splitting with 80 Processes per Group

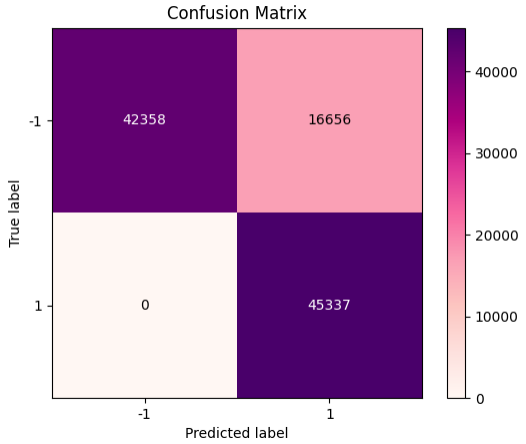

Figure 4.43: Confusion matrix for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

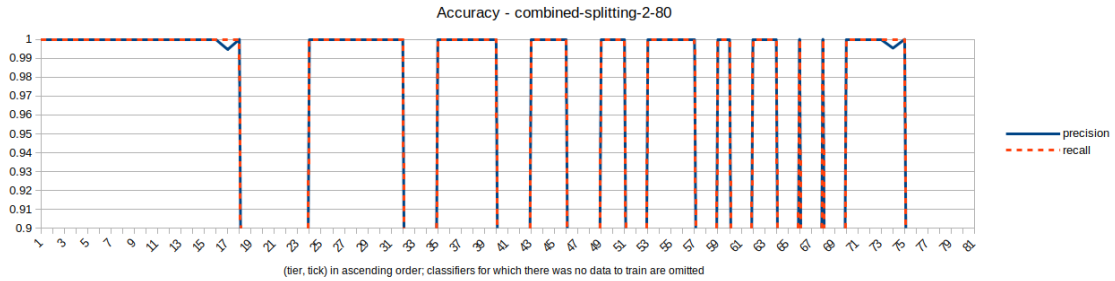

Figure 4.44: Accuracy of the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

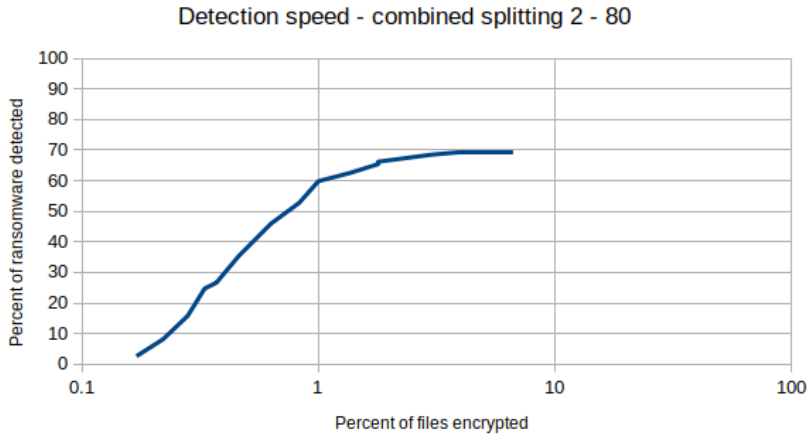

Figure 4.45: Detection speed for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

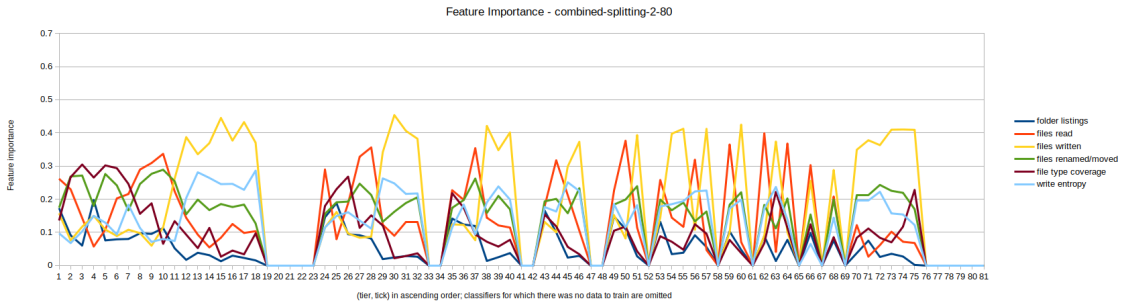

Figure 4.46: Feature importances for the 121 random forests for the process-centric half of the ShieldFS reimplementation when trained and tested with (DL,RN), (RD,WT) splitting for $K = 3$, 70%/30% train/test split, and 80 processes per group.

Recall	0.992
Precision	1
Specificity	1
Accuracy	0.999

Table 4.4: Results for the system-centric half of the ShieldFS reimplementation for $K = 3$ and 70%/30% train/test split.

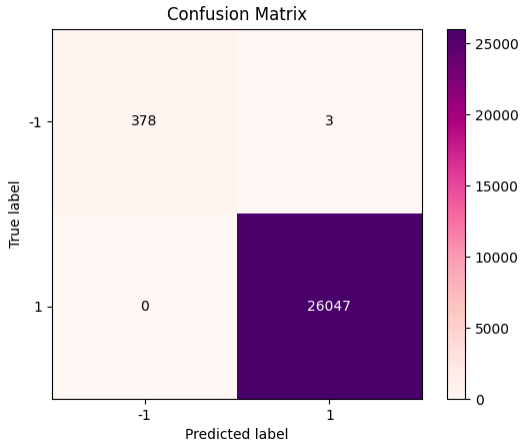

Figure 4.47: Confusion matrix for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

4.3 Straw Man Classifier

4.3.1 Testing Against the ShieldFS Dataset

The results of training and testing the system-centric half of the ShieldFS reimplementation, referred to as the *straw man classifier*, on the ShieldFS dataset are shown in Table 4.4, Figure 4.47, Figure 4.48, Figure 4.49, and Figure 4.50. The recall is very good and the precision is 100 percent. The results should be taken with a grain of salt because of the deficiencies in the ShieldFS ransomware dataset identified in §2.5.1, which may cause the straw man classifier to over-fit. Even with that possibility, we believe that the straw man classifier is likely to have very good real precision and recall.

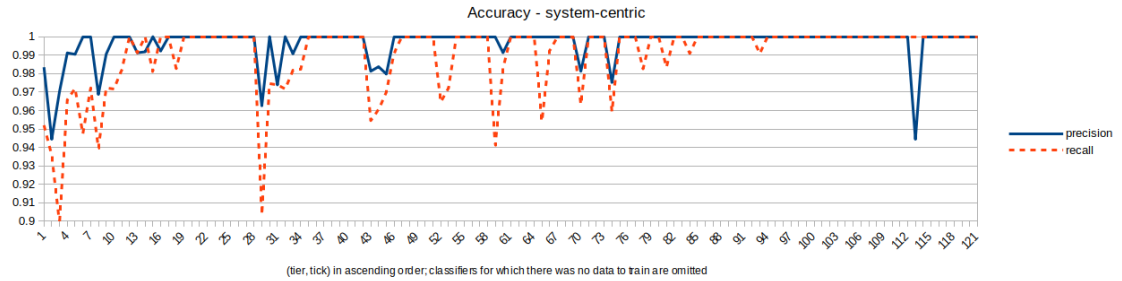

Figure 4.48: Accuracy of the 121 random forests for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

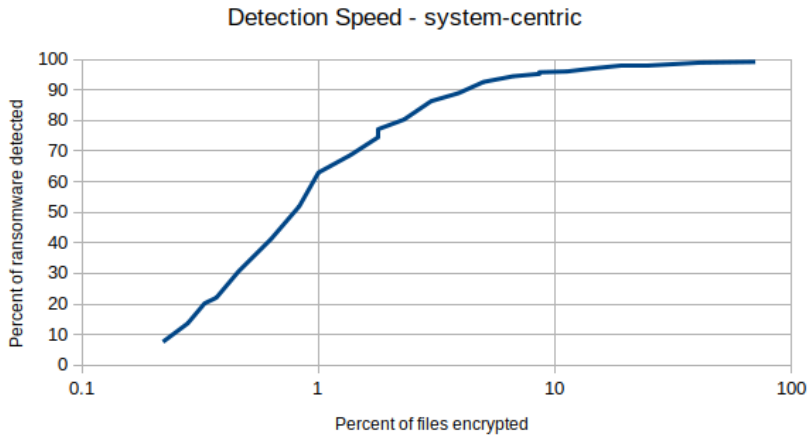

Figure 4.49: Detection speed for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

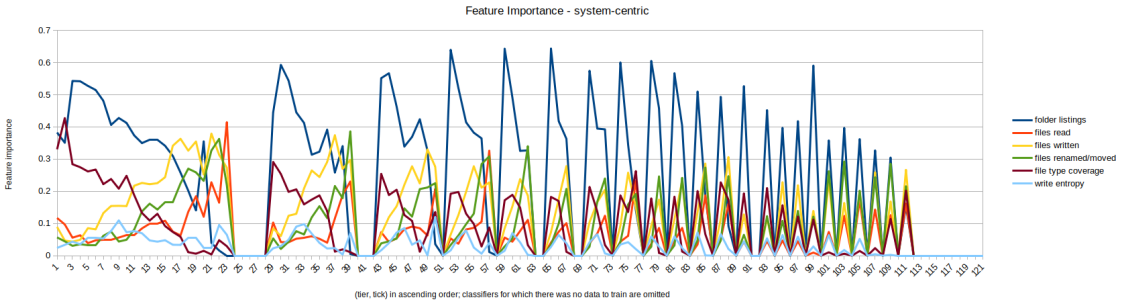

Figure 4.50: Feature importances for the 121 random forests for the system-centric half of the ShieldFS reimplementation using the ShieldFS dataset for $K = 3$ and 70%/30% train/test split.

Notably, the random forests for the final one or few ticks in each tier appear to have learned to always label everything as ransomware. This can be seen in Figure 4.50 because all of the feature importances drop to zero for those ticks. This means that if a benign program were to touch all or nearly all of the files on the machine, it would be labelled as ransomware. This also means that no such benign program was present in the ShieldFS benign dataset, because the precision of the straw man classifier was 1. This could be viewed as a deficiency in ShieldFS benign dataset, but the ShieldFS benign dataset was collected from real machines under real use.

The straw man classifier accomplishes Goal 1 of this thesis as described in §3.1. It correctly detects the presence of ransomware on the machine with high recall and precision. The straw man classifier somewhat accomplishes Goal 2, regarding the speed of detection. It detects the ransomware slightly more slowly than the process-centric model of the ShieldFS reimplementation; the comparison is between Figure 4.49 and Figure 4.19. Both are far slower than the detection speeds described in ShieldFS [3], but are quick enough to save a significant portion of the user's files. The straw man classifier does not accomplish Goal 3 of detecting which processes make up the ransomware, and does not accomplish Goal 4 regarding the speed of identifying the ransomware processes.

4.3.2 Testing Against Cerberus

The results of testing the straw man classifier against the Cerberus prototype developed by the The Naked Sun authors are shown in Figure 4.51. Cerberus is detected after encrypting 11.25% of files.

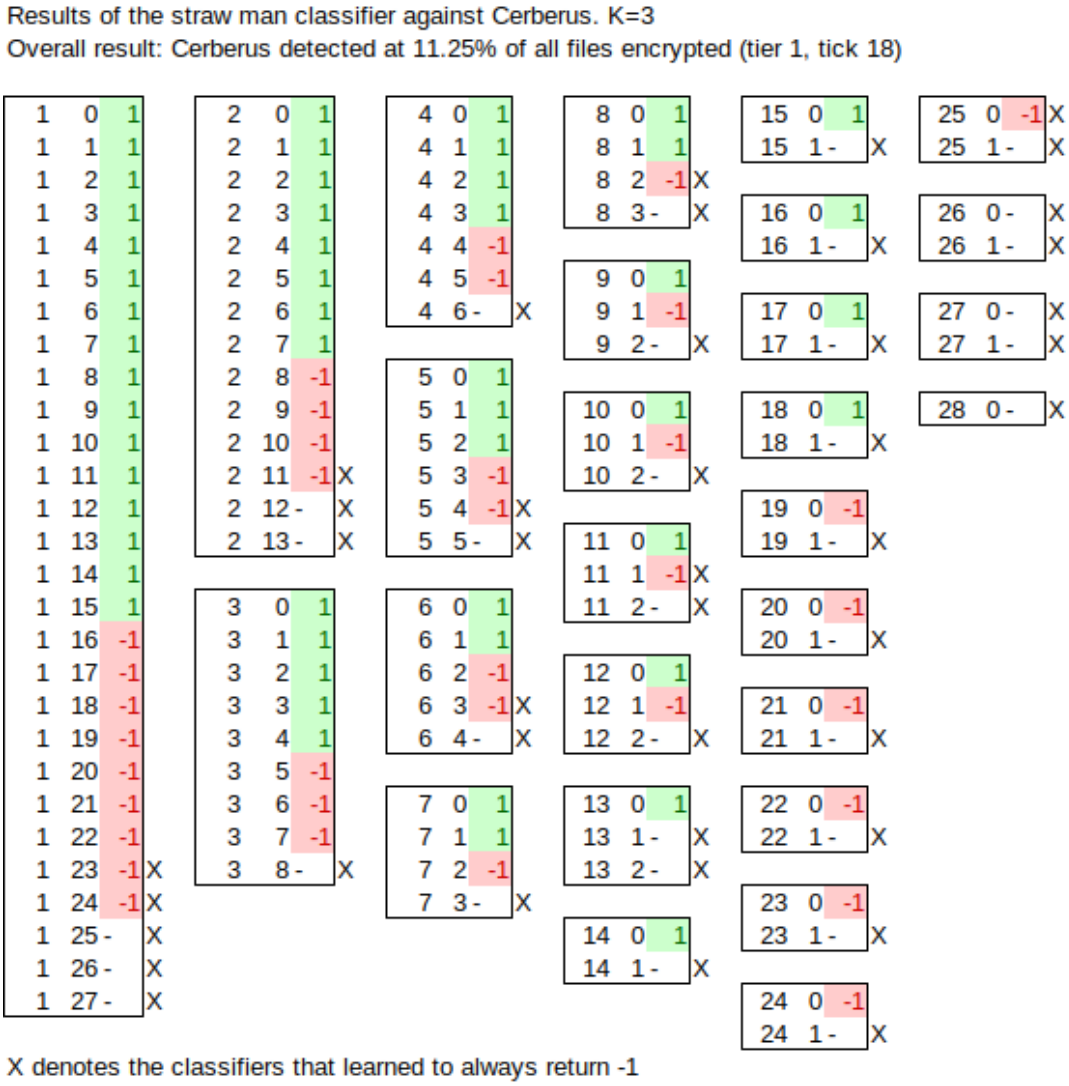

Figure 4.51: Labels output by the 121 random forests for system-centric half of the ShieldFS reimplementation when tested against Cerberus.

Recall	0.580
Precision	0.644
Specificity	0.995
Accuracy	0.989

Table 4.5: Accuracy of classifying files as attacked or not attacked using the ShieldFS dataset with a window size of the entire trace and with 70%/30% train/test split.

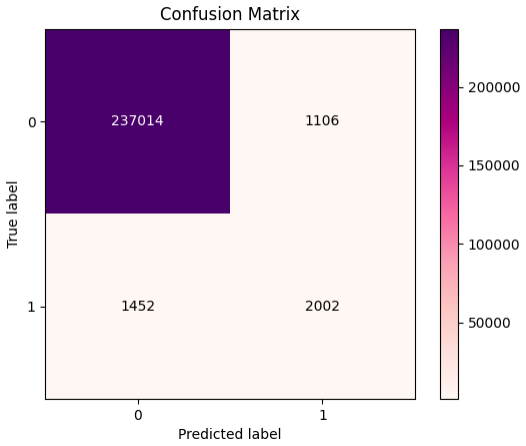

Figure 4.52: Confusion matrix for classifying files as attacked or not attacked using the ShieldFS dataset with a window size of the entire trace and with 70%/30% train/test split.

4.4 New Classifier

The new classifier was trained and tested with the ShieldFS dataset. Two window sizes were used: the entire trace and 10 seconds. The results of testing the new classifier using the ShieldFS dataset with a window size of the entire trace are displayed in Table 4.5, Figure 4.52, Figure 4.53, Table 4.6, and Figure 4.54. The results of testing the new classifier using the ShieldFS dataset with a window size of 10 seconds are displayed in Table 4.7, Figure 4.55, Figure 4.56, Table 4.8, Figure 4.57, Figure 4.58, and Figure 4.59.

If the new classifier, with the 10 second window, is used to merely detect the presence of ransomware rather than detect which processes are part of the ransomware,

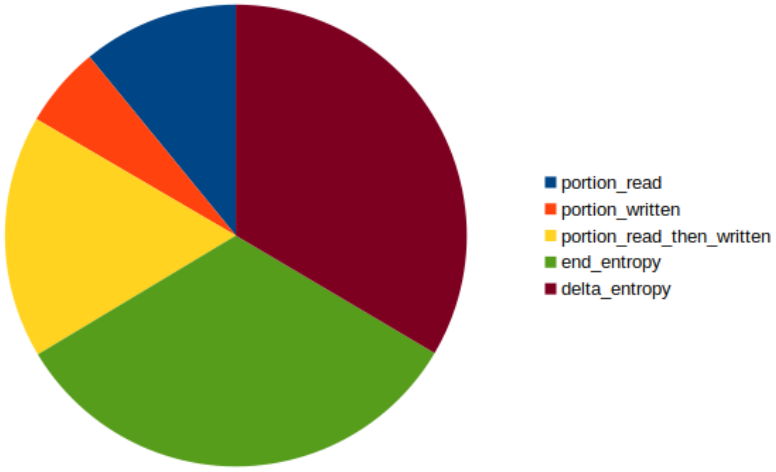

Figure 4.53: Feature importances for classifying files as attacked or not attacked using the ShieldFS dataset with a window size of the entire trace and with 70%/30% train/test split.

Recall	0.924
Precision	1
Specificity	1
Accuracy	0.9993

Table 4.6: Accuracy of detecting the mere presence of ransomware on the machine using the ShieldFS dataset with a window size of the entire trace and with 70%/30% train/test split.

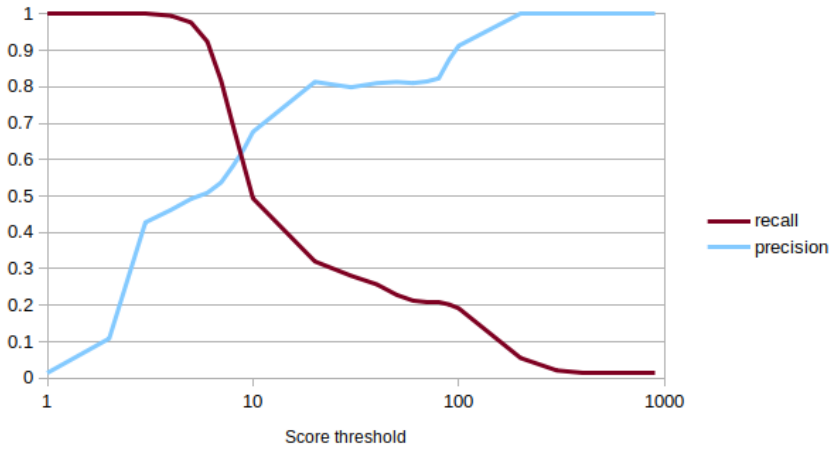

Figure 4.54: Accuracy of classifying processes as ransomware or benign using the ShieldFS dataset with a window size of the entire trace and with 70%/30% train/test split.

Recall	0.961
Precision	0.953
Specificity	0.983
Accuracy	0.977

Table 4.7: Accuracy of classifying files as attacked or not attacked using the ShieldFS dataset with a window size of 10 seconds and with 70%/30% train/test split.

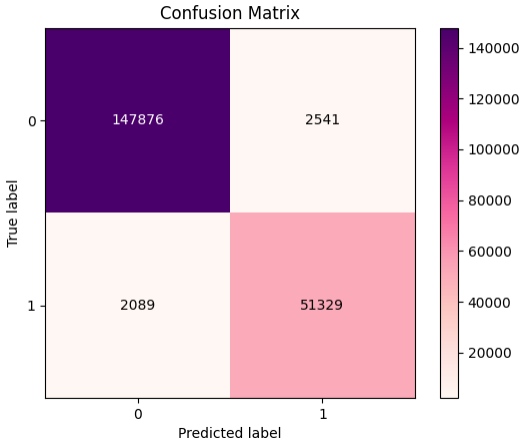

Figure 4.55: Confusion matrix for classifying files as attacked or not attacked using the ShieldFS dataset with a window size of 10 seconds and with 70%/30% train/test split.

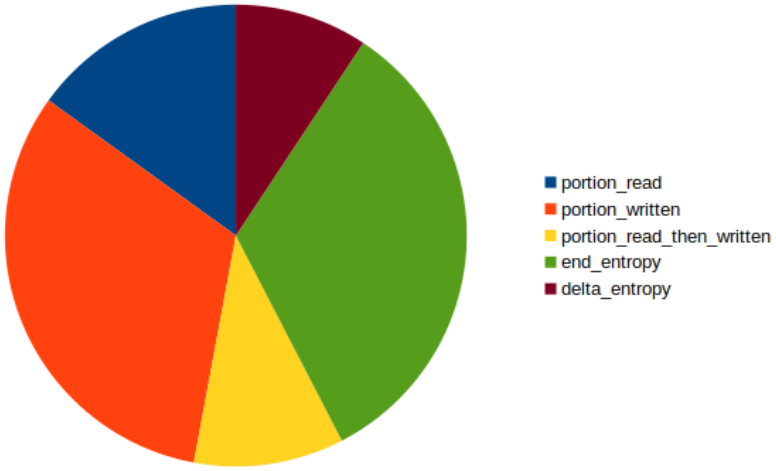

Figure 4.56: Feature importances for classifying files as attacked or not attacked the new classifier using the ShieldFS dataset with a window size of 10 seconds and with 70%/30% train/test split.

Recall	1
Precision	1
Specificity	1
Accuracy	1

Table 4.8: Accuracy of detecting the mere presence of ransomware on the machine using the ShieldFS dataset with a window size of 10 seconds and with 70%/30% train/test split.

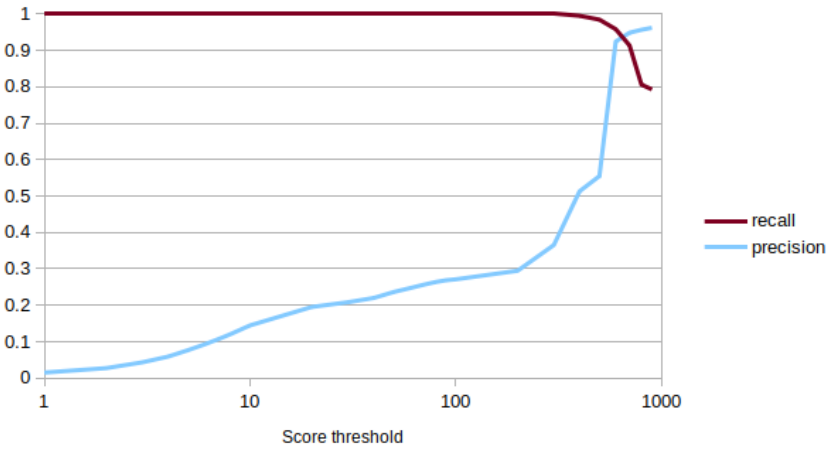

Figure 4.57: Accuracy of classifying processes as ransomware or benign using the ShieldFS dataset with a window size of 10 seconds and with 70%/30% train/test split.

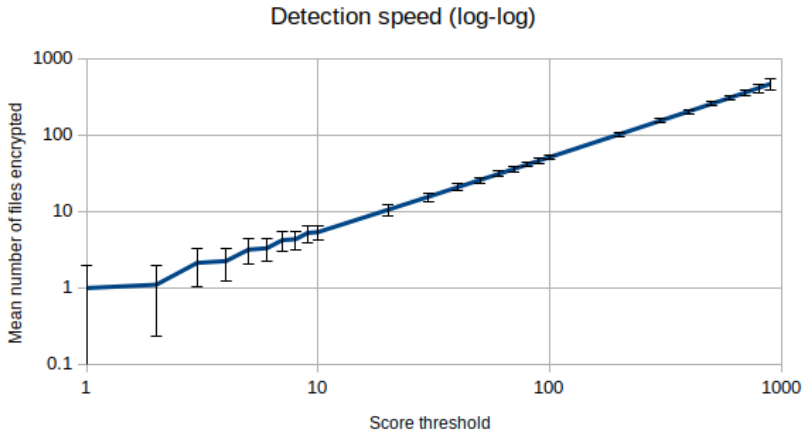

Figure 4.58: Detection speed of classifying processes as ransomware or benign using the ShieldFS dataset with a window size of 10 seconds and with 70%/30% train/test split.

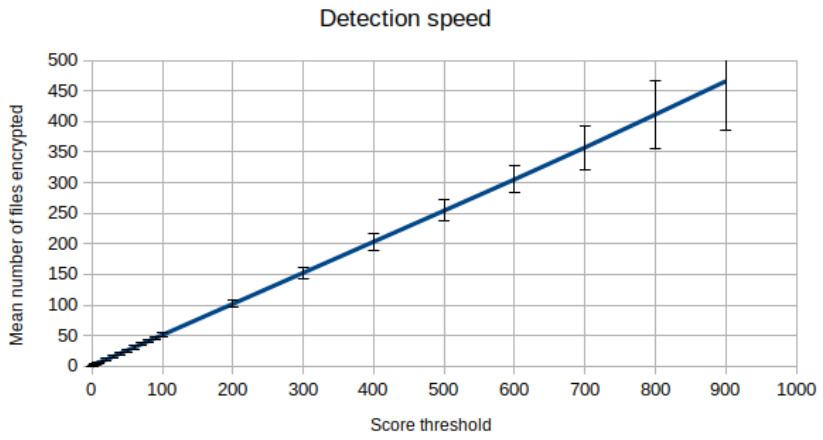

Figure 4.59: Detection speed of classifying processes as ransomware or benign using the ShieldFS dataset with a window size of 10 seconds and with 70%/30% train/test split.

then the new classifier achieves perfect recall and precision after about 10 files are encrypted. This detection speed is higher than that of the straw man classifier.

For detecting which processes are part of the ransomware, the new classifier with the 10 second window has excellent recall. However, the precision is very poor for low score thresholds. Using a score threshold of 600, the classifier has good recall and good precision. However, with such a high threshold, about 300 files are encrypted by the time the ransomware is detected. Since the detection speed for the straw man classifier is measured in percent of files encrypted rather than absolute number of files of encrypted, it is necessary to convert the 300 number to a percent in order to make a comparison. Considering the total number of files on each of the machines in the ShieldFS benign dataset, 300 ranges from 1.1 percent to 16.6 percent, with an average of 6.8 percent. The straw man classifier detects 94.5 percent of ransomware by the time that 6.8 percent of files are encrypted, which is shown in Figure 4.49. The new classifier detects 95.8 percent of ransomware by that same point (300 files).

By the reasoning of the preceding paragraph, the new classifier has a detection

speed that is as good as that of the straw man classifier. However, the straw man classifier can detect many ransomwares more quickly than 6.8 percent, whereas the new classifier, with a threshold of 600, will not detect any rasomware before 300 files are encrypted. Also, the straw man classifier detects further ransomwares as they encrypt more files, whereas the new classifier, with a threshold of 600, has a flat recall of 95.8 that does not improve further as even more files are encrypted.

The random forest that classifies files as attacked or not attacked has good precision and recall for the 10-second window. The poor precision of the process classification is not a result of poor precision of the file classifier, but rather results from the imprecision of the method of correlating from files to processes.

The new classifier accomplishes Goal 1 of this thesis as described in §3.1. It correctly detects the presence of ransomware on the machine with good recall and precision. The new classifier accomplishes Goal 2, regarding the speed of detection. It detects the presence of ransomware with perfect recall and precision after about 10 files are encrypted. The new classifier partially accomplishes Goal 3, detecting which processes are part of the ransomware; it accomplishes that goal with respect to ransomware that uses functional splitting, but not ransomware that uses process splitting. For functional splitting, the new classifier somewhat accomplishes Goal 4, regarding the speed of identifying the ransomware processes. It detects 95 percent of the ransomware processes about as quickly as the straw man classifier detects 95 percent of ransomware. However, it does not detect any processes before 300 files are encrypted because doing so would result in poor precision. For process splitting the new classifier does not accomplish Goal 4.

4.5 User Study

After two months of delay caused by the defect described in §3.7, the user study software is ready for use. We have a plan to move forward with code-signing the client software and deploying it on WPI-owned machines with the help of WPI's IT Services. The server is already deployed. This author plans to conclude his studies as a Master's student sooner than the user study can initiate.

Chapter 5

Discussion

The straw man classifier detects the presence of ransomware with high recall and precision. However, due to an isssue with the ShieldFS ransomware dataset described in §2.5.1, it may be slightly over-fitting. The new classifier detects the presence of ransomware with high recall and precision more quickly than the straw man classifier. Unlike the straw man classifier, the new classifier can also indicate which processes are part of the ransomware. The new classifier classifies processes with good recall, but it has good precision only after at least 300 files are encrypted by a given process. At that point, the recall of the straw man classifier and the new classifier are similar. The straw man classifier is theoretically immune to process and functional splitting, and detects the Cerberus prototype developed by the The Naked Sun authors. With regard to the detection of the presence of ransomware, the new classifier is theoretically immune to process and functional splitting. With regard to the classification of processes, the new classifier is theoretically immune to functional splitting, but it is not immune to process splitting.

The component of the new classifier that classifies files as attacked or not attacked has good recall and precision and is theoretically immune to both process

and functional splitting. The reason for requiring at least 300 files to be encrypted before labeling a process as ransomware is to have good precision with respect to the classification of processes, not files. The recall of the process classification is excellent at lower thresholds than 600, but the precision is poor. The vulnerability to process splitting is introduced at the correlation step where file classifications are translated into process classifications.

§3.1 lists the goals of this thesis. The straw man classifier accomplishes Goal 1 and somewhat accomplishes Goal 2. The new classifier accomplishes Goal 1 and Goal 2, and partially accomplishes Goal 3.

We believe that a better method of correlating the results of the file classifier to classify processes is needed. Such a method should maintain the immunity to process splitting and should have better precision when less than 300 files have been encrypted.

Even if the precision of the process classification is not improved, the new classifier may still be of practical use. It can detect the mere presence of ransomware with high recall, precision, and speed. It can detect which processes are part of the ransomware with high recall and speed, but low precision, or with high recall and precision, but low speed. It could be used to detect the presence of ransomware and then to provide a list of candidate ransomware processes which could then be analyzed with some other system.

All software needed for the user study has been created and tested. The user study has also passed the IRB review and should be able to initiate quickly. The user study should collect a new dataset that will not have the shortcomings of the ShieldFS dataset identified in §2.5.1 and §4.3 and that will include the additional data of the sizes of files and EnCoD [33] labels on operations.

Chapter 6

Conclusion

The multiprocess evasion techniques described in The Naked Sun allow ransomware to evade state-of-the-art ransomware detectors. We replicated some of the results of The Naked Sun and of ShieldFS. We then developed two classifiers that detect the presence of ransomware that uses the multiprocess evasion techniques, the first being based on the system-centric portion of the ShieldFS detector, and the second being based on classifying files as attacked or not attacked. Both detect the presence of such ransomware with high precision and recall. The second classifier can also determine which processes are part of the ransomware in the face of the functional splitting technique, but not of the process splitting technique. Determining which processes are part of the ransomware requires many files to be encrypted in order to have good precision. We also prepared for a user study to collect a new dataset, developing the necessary client and server software.

Future work involves detecting which processes are part of the ransomware even in the face of process splitting and with a better precision/speed trade-off than the current detector. Future work should also execute the user study and collect a new ransomware dataset.

References

[1] F. De Gaspari, D. Hitaj, G. Pagnotta, L. De Carli, and L. V. Mancini, "The Naked Sun: Malicious Cooperation Between Benign-Looking Processes," tech. rep., arXiv:1911.02423 [cs.CR], Nov. 2019.

[2] F. De Gaspari, D. Hitaj, G. Pagnotta, L. De Carli, and L. V. Mancini, "The Naked Sun: Malicious Cooperation Between Benign-Looking Processes," in *ACNS 2020: Applied Cryptography and Network Security*, vol. 12147 of *Lecture Notes in Computer Science*, (Cham, Switzerland), pp. 254–274, Springer International Publishing, 2020.

[3] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "ShieldFS: a self-healing, ransomware-aware filesystem," in *ACSAC '16: Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, (New York, NY, United States), pp. 336–347, Association for Computing Machinery, 2016.

[4] S. Mehnaz, A. Mudgerikar, and E. Bertino, "RWGuard: A Real-Time Detection System Against Cryptographic Ransomware," in *RAID 2018: Research in Attacks, Intrusions, and Defenses*, vol. 11050 of *Lecture Notes in Computer Science*, (Cham, Switzerland), pp. 114–136, Springer International Publishing, 2018.

[5] Malwarebytes, "Malwarebytes Anti-Ransomware for Business."

[6] D. Goodin, "New Android ransomware locks out victims by changing lock screen PIN," Sept. 2015.

[7] C. Cimpanu, "Android Ransomware Infects LG Smart TV," Dec. 2016.

[8] Coveware, "Ransomware Payments Decline in Q4 2020," Feb. 2021.

[9] Sophos, "The State of Ransomware 2020," tech. rep., May 2020.

[10] Cybersecurity and Infrastructure Security Agency, "CISA Launches Campaign to Reduce the Risk of Ransomware." Jan. 2021.

[11] Coveware, "Ransomware Attacks Fracture Between Enterprise and Ransomware-as-a-Service in Q2 as Demands Increase," Aug. 2020.

[12] D. Goodin, "North Korea's Lazarus brings state-sponsored hacking approach to ransomware," July 2020.

[13] S. Kumar and E. H. Spafford, "A generic virus scanner for C++," in *Proceedings of the Computer Security Applications Conference*, pp. 210–219, 1992.

[14] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 421–430, 2007.

[15] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The Hidden Malware," *IEEE Security Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[16] J. Salvio, "Dissecting Mamba, the Disk-Encrypting Ransomware," Sept. 2016.

[17] A. Viviano, D. Coulter, C. Gregg, and N. Bazan, "I/O request packets," Apr. 2017.

[18] A. Viviano and N. Bazan, "What is a driver?," Apr. 2017.

[19] L. W. Hollasch and D. Coulter, "Filtering IRPs and Fast I/O," Apr. 2017.

[20] L. W. Hollasch, "Filter Manager Concepts," Feb. 2020.

[21] G. Pagnotta, "IRPLogger." GitHub repository, July 2020.

[22] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 757–772, USENIX Association, Aug. 2016.

[23] A. Kharraz and E. Kirda, "Redemption: Real-Time Protection Against Ransomware at End-Hosts," in *RAID 2017: Research in Attacks, Intrusions, and Defenses*, vol. 10453 of *Lecture Notes in Computer Science*, (Cham, Switzerland), pp. 98–119, Springer International Publishing, 2017.

[24] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, "CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 303–312, June 2016.

[25] J. Lee, J. Lee, and J. Hong, "How to Make Efficient Decoy Files for Ransomware Detection?," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '17, (New York, NY, USA), pp. 208–212, Association for Computing Machinery, 2017.

[26] C. Moore, "Detecting Ransomware with Honeypot Techniques," in *2016 Cybersecurity and Cyberforensics Conference (CCC)*, pp. 77–81, Aug. 2016.

[27] R. Moussaileb, B. Bouget, A. Palisse, H. Le Bouder, N. Cuppens, and J.-L. Lanet, "Ransomware's Early Mitigation Mechanisms," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, (New York, NY, USA), Association for Computing Machinery, 2018.

[28] J. Gómez-Hernández, L. Álvarez González, and P. García-Teodoro, "R-Locker: Thwarting ransomware action through a honeyfile-based approach," *Computers & Security*, vol. 73, pp. 389–398, Mar. 2018.

[29] Z. A. Genç, G. Lenzini, and D. Sgandurra, "On Deception-Based Protection Against Cryptographic Ransomware," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, eds.), (Cham, Switzerland), pp. 219–239, Springer International Publishing, June 2019.

[30] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast Memory-Efficient Anomaly Detection in Streaming Heterogeneous Graphs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 1035–1044, Association for Computing Machinery, 2016.

[31] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan, "HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1137–1152, May 2019.

[32] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats," in *Proceedings 2020 Network and Distributed System Security Symposium*, 2020.

[33] F. De Gaspari, D. Hitaj, G. Pagnotta, L. De Carli, and L. V. Mancini, "EnCoD: Distinguishing Compressed and Encrypted File Fragments," tech. rep., arXiv:2010.07754 [cs.CR], Oct. 2020.

[34] Microsoft, "MessageBox function (winuser.h)," Dec. 2018.

[35] Microsoft, "Microsoft Visual Studio Installer Projects."

[36] Microsoft, "InstallHinfSectionW function (setupapi.h)," Dec. 2018.