ABSTRACT

Title of dissertation: SYSTEMATIC ANALYSIS OF ADVERSARIES' EXPLOITATIONS OF THE END-HOST

Erin Avllazagaj
Doctor of Philosophy, 2023

Dissertation directed by: Professor Tudor Dumitraș
Professor Yonghwi Kwon

Department of Electrical and Computer Engineering

In the pipeline of a cyber attack, the malicious actor will first gain a foothold in the target system through a malware. The malware detection is still a challenging problem, as the malware authors are constantly evolving their techniques to evade detection. Therefore, it is important for us to understand why that is the case and what can the defenders do to improve the detection of the malware. In this thesis, I explore the behavior of the malware in the real users' machines and how it changes across different executions. I show that the malware exhibits more variability than benign samples and that certain actions are often more prone to variability than others. This is the first study that quantitatively analyzes the behavior of the malware in the wild.

I leverage an observation from the first project, where variability in the malware samples happens due to running privilege escalation exploits. The variability

in behavior is due to the fact that the malware sometimes runs in non-privileged mode and tries to run an exploit to escalate its privileges. For these reasons, I propose a new methodology to systematically discover sensitive memory corruption targets that cause privilege escalation.

At last, I explore the sensitive memory corruption targets in the Linux kernel. Specifically, I propose a methodology to systematically discover sensitive fields in the Linux kernel that, when corrupted, lead the system into an exploitable state. This system, called SCAVY, is based on a novel definition of the exploitable state that allows the attacker to read and write into files and memory locations that they would normally. SCAVY explores the exploitable states based on the threat model of a local unprivileged attacker with the ability to issue system calls and with the capability to read/write into a limited location in the kernel memory.

The framework revealed that there are 17 sensitive fields across 12 Linux kernel `C structs` that, when overwritten with the correct value, lead the system into an exploitable state. In this definition, unlike prior work, I consider the system to be in an exploitable state when the weird machine allows the attacker to *read* and/or *write* into files and memory locations that they would normally not be able to. This state can be used to write into sensitive files such as `/etc/passwd` where the exploit author can create a new root account on the vulnerable host and log in as that. Additionally, if the attacker can read unreadable files such as `/etc/shadow` they can leak passwords of root accounts, de-hash them and log in as the root account.

I utilize these targets to develop 6 exploits for 5 CVE vulnerabilities. I also demonstrated the severity of these fields and the applicability of the exploitable state

by exploiting *CVE-2022-27666*. I overwrote the *f_mapping* pointer in `struct file` and caused a write into */etc/passwd*. Unlike the original exploit, ours didn't need to break KASLR, modify global variables or require support of `FUSE-fs` from the vulnerable host. This makes our methodology more extensible and more stable, since the exploit requires fewer corruption in the kernel memory and it doesn't rely on the need to have the addresses of the kernel's symbols for calculating the KASLR offset. Additionally, our exploit doesn't modify global variables, which makes it more stable and less likely to crash the kernel, during its runtime. Our findings show that new memory corruption targets can change the security implications of vulnerabilities, urging researchers to proactively discover memory corruption targets.

AUTOMATIC EXPLORATION OF THE VULNERABILITY LANDSCAPE IN LINUX KERNEL

by

Erin Avllazagaj

Under the Guidance of Professor Tudor Dumitraș &
Professor Yonghwi Kwon

Submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:
Professor Tudor Dumitraș, Chair/Advisor
Professor Yonghwi Kwon, Co-Chair/Co-Advisor
Professor Nirupam Roy
Professor Dana Dachman-Soled
Professor David Van Horn, Dean's Representative

PUBLICATIONS

The dissertation is partly based on work that has been published during the author's PhD in the following paper(s):

1. **Avllazagaj, E.**, Kwon, Y., & Dumitraș, T. (2024). SCAVY: Automated Discovery of Memory Corruption Targets in Linux Kernel for Privilege Escalation. In 33rd USENIX Security Symposium (USENIX Security 24).

2. **Avllazagaj, E.**, Zhu, Z., Bilge, L., Balzarotti, D., & Dumitraș, T. (2021). When malware changed its mind: An empirical study of variable program behaviors in the real world. In 30th USENIX Security Symposium (USENIX Security 21) (pp. 3487-3504). **(1^{st} place winner at CSAW applied research competition)**

3. Garg, N., Shahid, I., **Avllazagaj, E.**, Hill, J., Han, J., & Roy, N. (2023, February). Thermware: Toward side-channel defense for tiny iot devices. In Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications (pp. 81-88).

4. Suciu O., **Avllazagaj, E.**, & Dumitraș, T. (2019) Secret Life of Pwns: Characterizing and Predicting Exploit Weaponization. In Conference on Applied Machine Learning in Information Security (CAMLIS) 2019.

ACKNOWLEDGEMENTS

Looking back, this research journey has been a sequence of decisions in the control flow graph of my life. My PhD path starts from an email my undergraduate advisor, Prof. Erman Ayday, received in the spring of 2017. Prof. Tudor Dumitraș email was an invitation for bright minds from Bilkent University to join his research group for a summer internship. In hindsight, I am very grateful to have received this opportunity and I'm grateful to Prof. Ayday for encouraging me to apply.

During the internship I met brilliant people that taught me the importance of collaboration and the value of a supportive research environment. I am grateful to my peers, Cip, Florina, Mirac for their friendship and fun times we had together. You made the post-research hours in the lab enjoyable and memorable. I am also grateful to my mentor, Dr. Octavian Suciu, for his guidance and support during the internship. I learned a lot from him and I am thankful for the guidance to researching and questioning evey step I take. Finally, I am grateful to Prof. Tudor Dumitraș for accepting me as an intern and for his unrelenting support and availability during the internship, as well as believing in me and offering me a PhD position.

Thus, *Return of the Intern* became the next season of my life, after receiving the lightsaber from Yigitcan. It started earlier than expected, when I was tasked to go to *EURECOM* for a month in July 2017. There I met brilliant researchers and hackers and learned their ways of thinking and working. Thanks to my mentors, Dr. Leyla Bilge and Dr. Davide Balzarotti, for helping me throught my first ever visit

in France and understanding the BASH dataset. On my return, I was welcomed by the Tudor and the group in USENIX 2018. I am thankful to attend the conference and attend the 2 papers presented by our group (congratulations to Octavian and Doowon).

During my first few years of PhD, pre-COVID, I had the opportunity to meet and learn from peers in my office. I want to thank my peers, especially Lambros Mertzanis for biking at freezing temperatures to study for the qualifying exam and helping me with brushing up on probability. Thank you to all my research group members, Ziyun, Sanghyun, Doowon, BumJun, Octavian and Yigitcan for their friendship and the good times. Thank you Doowon for being your trolling self and for always being a good sport. I will keep sending you reels and screenshots of UMD Alerts from the GTA server called Baltimore Avenue. Thank you Sanghyun for the late night coffee breaks and for giving me the encouragement I needed to keep going on with my research evenwhen all felt futile. Be wary of any research ping you might receive at 4AM (local time).

I am thankful to the interns I met during my PhD, some of which are following this same academic path. Thank you Georgios, Tasos, Simge and especially Michi, for his great work even during the pandemic. I wish your research to be groundbreaking and your papers to be accepted in top-tier conferences, because you deserve it. May we meet up again sometime at the dining hall and have pizza (with mayo), mongolian grill shrimp and gyros.

I am thankful to all the mentors in my PhD journey. Many thanks go to my advisor Prof. Tudor Dumitraș for his guidance and support during my PhD. Thank

you for keeping me on track with the question, "*What are the contributions? What are the next steps?*". I learned from you the importance of looking at the big picture and questioning the need and use of each experiment on the overall research project. Thank you for being awesome and I hope I can join you on bouldering in the future. Thanks to my co-advisor, Prof. Yonghwi Kwon for your help in teaching me to better present a paper to my reviewers and to simplify my writing. Thank you also for being availabile until the last minute before the submission deadline to fix every little detail in the paper. At last, many special thanks go to Prof. Nirupam Roy for teaching me IoT security. The class is a masterpiece in terms of content and delivery and I am grateful for the opportunity to attend it and have an accepted paper on the class project, Thermware. Thank to Nakul too for being availabile to help with the project.

At last I am thankful to my family for their support and encouragement during my PhD. Thank you to my parents for always calling me and asking how I am doing and for their support in my academic journey, especially when I was there duing the pandemic. I'm grateful to my dad for making me breakfast at bringing snacks when I was working. I'm grateful to my mom for cooking and doing laundry so that I would have all the time to focus on my work. I'm grateful to my brother for joining me at the gym and keeping up with me. I'm grateful to my uncle Rudi Domi for believing in me and for his support and encouragement. At last, I am grateful to my grandparents, Mustafa and Nashifer for always asking me how I am doing and for their support and encouragement. I'm glad you feel proud of me and I hope you are there to see me succeed.

Finally, I am thankful to my partner, Bao Ngoc (aka. Jamie) Dinh, for her support (especially for the dinners when I was on time crunch) and for the late night work sessions we had together. I'm looking forward to the next chapter in our life together and I hope we can make it a great one.

Erin Avllazagaj

August, 2024

Maryland, USA

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

A step in the cyber attack involves establishing foothold in the target host through a malware. The malware detection is still a challenging problem, as the malware authors are constantly evolving their techniques to evade detection. In this thesis I explore the behavior of the malware in the real users' machines and how it changes across different executions. In a collaboration with an AV vendor, we collected 7.6M execution traces from 5.4M real hosts from 113 countries. In this joint project, we analyze how these malware evaded the AV vendor's detections and how the behavior variability affects detectors proposed in prior works. Additionally, we highlight the actions that are more prone to variability. This is the first study that quantitatively analyzes the behavior of the malware in the wild.

At last, I leverage an observation from the first project, where variability in the malware samples happens due to running privilege escalation exploits. I observed in the first work, a local unprivileged attacker running the malware in the victim's machine will attempt to escalate their privileges only if it's not already running as admin, which is a reason for malware variability. When studying the severity of a kernel bug, prior work considers exploits as severe whenever an attacker can run their code in kernel context. However, in most cases we found that the

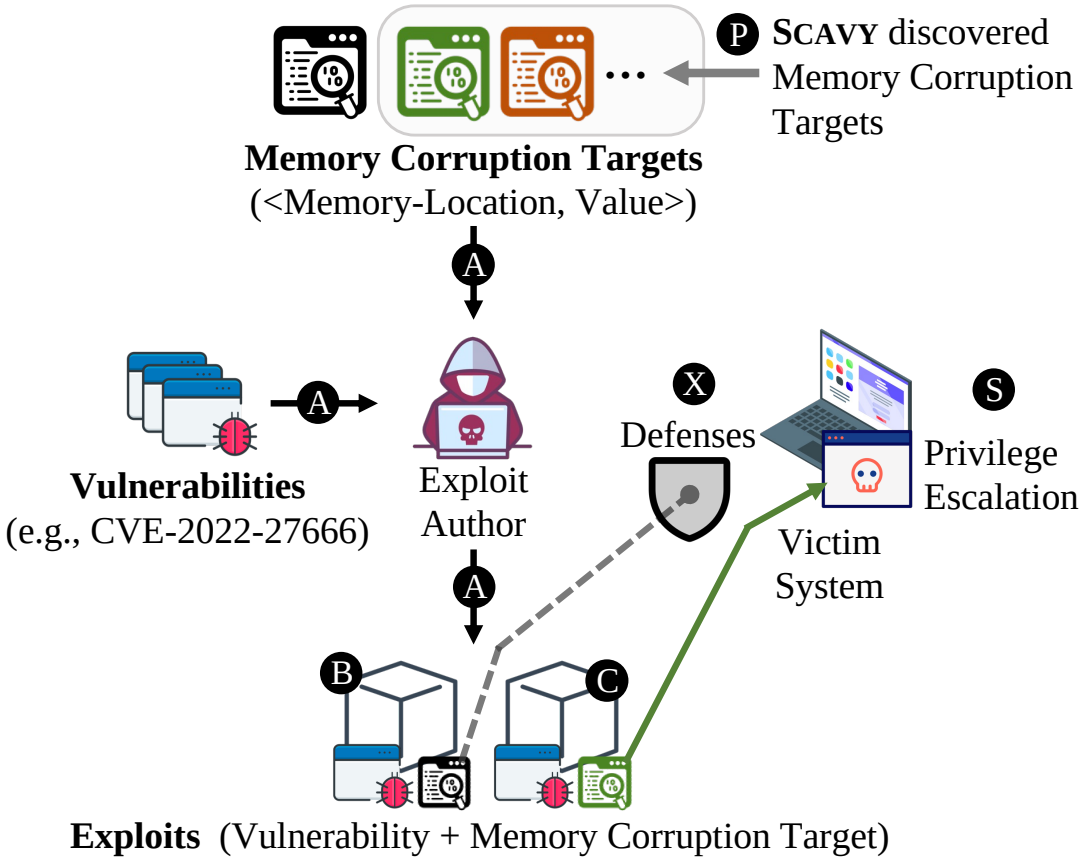

Figure 1.1: Memory Targets in Exploit

attacker simply needs to escalate their privileges to admin to achieve their goal. This escalation is often overlooked in the severity assessment of a kernel bug and is not systematically explored. I look into this problem by abstracting away the specific details of an exploit by emulating them as memory corruption capabilities that an attacker achieves through a vulnerability. From this point I study the effects that the memory corruptions have when applied to different memory targets in the kernel space and how they can be used to achieve privilege escalation. In the last project, I propose a new methodology to systematically discover sensitive memory corruption targets that cause privilege escalation.

For example, Figure 1.1 shows an exploit created by combining a vulnerability and a memory target (Ⓐ), resulting in multiple combinations (or exploits) such as Ⓑ and Ⓒ. Assume that a privilege escalation exploit (Ⓑ in Figure 1.1) seeks to corrupt *a function pointer* in a kernel data structure to execute a malicious payload that escalates the privilege. Unfortunately, the attacker in this exploit utilizes an *unnecessarily strong* capability, leading to a rather *obvious* exploitation method (i.e., control flow hijacking) for privilege escalation which is prevented by popular defenses such as CFI (Ⓧ). A *new memory target* can enable the attacker to escalate the privilege (Ⓒ) without being detected (Ⓢ). For example, corrupting other memory targets such as `username`, `inode` numbers, or the `task_struct::addr_limit` field, can accomplish privilege escalation. In particular, the `addr_limit` field [1] was famously employed in 2019 by the NSO group to install the Pegasus spyware on Android devices [2]. The vulnerability had been found by Syzkaller in 2017, but it was not patched in many released devices for many years, as its security implications were unknown [2]. As a result, in 2020, the `addr_limit` was used in over 40% of the Android kernel exploits in the wild [3]. This results in the removal of the field from the Linux kernel to prevent such attacks [4]. As such, the discovery of new memory targets enables the exploitation of many vulnerabilities including those that are *previously considered unexploitable or not critical* [5]. Moreover, new types of memory targets can enable exploits to evade existing defenses focusing on the popular memory targets [6–10]. Discovering memory targets is also beneficial in a defensive context, as several techniques (e.g., freelist pointer obfuscation [11]) can secure these kernel objects [12].

Despite its importance, finding memory targets has been challenging and typically done *manually*, relying on expertise in the Linux kernel code base. The prior research on automating kernel exploit generation [13, 14] has focused on exploring specific vulnerabilities with limited types of memory targets (e.g., function pointers or reference-count fields). Consequently, the fields of Linux kernel data structures that privilege-escalation exploits can target have not been systematically researched. For instance, among the 6,582 structures in the Linux kernel version 5.15.80, only 746 (11.3%) of them contain either function pointers (142; 2.1%) or pointers to structures containing function pointers (604; 9.2%) that are considered security-sensitive and have been actively addressed by previous work. The remaining 88.7% represent an *unexplored attack surface*. Moreover, even for the previously addressed 11.3%, an additional thorough search is desirable as [13, 14] only examined reachable memory areas from existing proof-of-concept (PoC) code and [15] focuses on a few kernel objects that induced crashes during the analysis.

Chapter 2: Thesis Objectives

In this thesis, I present a large scale study of malware behavior in the wild. I analyze the behavior of malware, PUP and benign samples in the wild, using execution traces collected from 5.4M real hosts from around the world. I show that malware exhibits different levels of variability in different actions and evaluate prior

state-of-the-art detectors and classifiers on the dataset to highlight the issues with using sandbox data in evaluations. At last, I leverage an observation from the first project, where variability in the malware samples happens due to running privilege escalation exploits. The variability in behavior is due to the fact that the malware sometimes runs in non-privileged mode and tries to run an exploit to escalate its privileges.

For these reasons, I propose a new methodology to systematically discover sensitive memory corruption targets that cause privilege escalation, further challenging the definition of exploitability from prior works. SCAVY[1], a framework for systematically discovering *memory targets* in the Linux kernel for privilege escalation (Ⓟ in Figure 1.1). SCAVY searches for *broader types* of memory targets, beyond the pointer-type memory targets that existing techniques focus on [16–19]. In particular, SCAVY aims to discover new diverse types of targets that can impact (or enable) many existing and unknown vulnerabilities. During the analysis of the kernel structures and memory areas for the memory targets, SCAVY encounters two challenges: (1) examining a large number of potential memory targets and (2) lack of oracles that can detect diverse forms of potential privilege escalations. SCAVY handles the two challenges by leveraging a differential analysis-based multi-execution reasoning, that are more efficient than existing techniques rely on symbolic and taint analysis. Specifically, it runs multiple executions with and without memory corruption and checks the program states of the executions in terms of the accessibility of security-sensitive resources. The results are analyzed to guide the search process and detect

[1]SCAVY is an abbreviation for 'Scavenger.'

privilege escalations.

SCAVY generated 955 PoC exploits that can potentially escalate privileges, by corrupting 275 unique fields in 86 kernel structures. From the PoC exploits from `CVE-2022-27666`, we create *two* fully functional privilege escalation exploits with new SCAVY identified memory targets (one of which explained in subsection 4.2.1). Our exploits do not require bypassing popular kernel defenses (e.g., KASLR, SMEP/SMAP, and CFI), unlike the original exploits that had to handle them.

2.1 Structure of the Thesis

TODO: Update this section after writing the thesis.

The thesis is organized in the following way. section 4.1 presents the problem statement and the challenges in discovering memory targets. In section 4.2 I provide the motivation for SCAVY and discuss its utility in discovering new memory targets. In section 4.3, I review the related works.

Chapter 3: Malware behavior analysis in the wild

In this chaper I present our analysis of malware behavior in the end hosts. We collected a dataset of 7.6M execution traces, recorded in 5.4M real hosts from 113 countries. Our results show that malware exhibits more variability across machines. We tested state-of-the-art malware detection techniques and found that they are not effective in detecting malware in the real world, calling into question the validity of results derived sandbox data. In the following sections, I discuss the problem statement (section 3.1), the collected dataset (section 3.2), measurements of variability (section 3.3) and a study on the invariants usable for detection (section 3.4). I conclude this chapter with a a study on the impact in current detection and clustering methods(section 4.8).

3.1 Problem and Methodology

The main goal of malware analysis is to identify and characterize the behavior of unknown samples such that behavioral indicators that are specific to a malware family could be used for malware detection or classification. Because the behavior of executables could vary depending on when, where and at what setting it is executed, part of the behavior for any given program is transient in nature.

In our dataset, we observed that some executions of the Ramnit worm [20], result in the creation of a large number of mutexes. The reason is that the worm uses a privilege escalation exploit, which creates a lot of mutexes, only if it is executed in user-mode on a vulnerable version of Windows 7. If instead Ramnit is executed with admin privileges or within a different Windows version, the malware would not perform the exploit. If an analyst, or an automated system, created a signature by looking at the behavior collected on Windows 7 (a popular choice by many malware analysis sandboxes), those mutex creations could be used for constructing the signature. However, these actions would only appear in a fraction of end user machines, thus resulting in a poor detection coverage.

To mitigate this problem and identify truly invariant parts of malware behavior, it is important to collect malware executions across multiple machines, as suggested by Rossow et al. [21] and over time, as suggested by Pendlebury et al. [22]. However, prior works does not make concrete recommendations for the most optimal set up (e.g, the optimal re-execution interval, the number of different machines, the number of different OSes, etc.) that allows those invariant parts to be identified accurately. Our goal is to fill this gap in the state-of-the-art.

Despite these very time-consuming therefore costly suggestions, the industry practitioners often choose to aggregate behavior of different samples of a family for signature generation [23, 24]. However, the majority of malicious samples cannot be mapped to a known family (malware with generic labels are 1.3 times more common than those that belong to a well-defined family) [25], making it impossible to perform such an aggregation.

To shed light on the magnitude of this problem, we analyze $7.6M$ executions out of which $3.1M$ belong to malicious and unwanted programs and the rest to benign. In total, the executions of each sample span at least 10 machines, while 45% appear at least 1 week from the sample's first appearance. This measurement, the first of its kind, allows us to assess the amount of behavior variability in the wild, and to study the minimum number of experiments required to rule out transient behaviors and derive signatures that achieve the highest coverage on end hosts, filling a crucial gap in the state-of-knowledge about the most optimal execution configurations for signature generation.

3.1.1 Measuring Variability

We describe the behavior of a sample through its interactions with the host Operating System. Because a semantic interaction, such creating a new file or spawning an OS process, may be accomplished with various system or API calls, and the calls differ across OS versions, we abstract these interactions as *actions*. Our actions model high-level operations, such as process injection, file creation, or the modification of a registry key; we report all the action types analyzed in Section 3.2. An action may have one or several *parameters* to specify the target that the action is operating on (e.g. the registry key being modified), as well as the actual value it writes or modifies (e.g. the value written in the registry). An *execution trace* for a sample consists of a sequence of actions and the corresponding parameters. The traces captured by malware detectors based on both system calls [26, 27] and native

API calls [24, 28–31] can be mapped to action-based execution traces.

We measure variability at two levels of granularity. First, we count the actions in an execution trace and compute the *action variability*. We maintain separate counts for each action type, as well as for all the actions taken together. We then compare these counts across all the execution traces of a sample, using several measures of variability as described below. This provides a conservative assessment of variability, indicating for example when a sample creates one file on a host and two on another. We report *how much* action variability we observe, *which action types* account for most of the variability, and how these the variability changes across *space* (a sample executing on different hosts in the same week) and *time* (a sample executing on the same host in different weeks). We also compare the action variability in malware, PUP, and benign samples.

Second, we compare the action parameters coming from different execution traces of the same sample, using measures of set similarity. This *parameter variability* allows us to identify differences among executions when the number of actions remain the same, for instance when a sample creates a file with different names on each host. This comparison provides further insight into the semantics of the variable actions; for instance, we identify which parameter parts (e.g., the filename vs the directory path) differ among different executions.

Measuring action variability. The action counts coming from different execution traces of a sample form an empirical distribution. We can characterize this distribution using various measures of location (e.g. mean, median, mode) and spread

(e.g. variance, standard deviation, median absolute deviation, interquartile range); we are interested in the latter when assessing action variability. The main challenge in selecting a statistical measure of spread is to avoid drawing incorrect conclusions because of outliers in the distribution.

We illustrate this challenge by showing how different variability measures perform over the executions of one sample of AutoPico, a Windows piracy software. Usually the sample creates four files when executed: two `log` files, one `dll` and one `.sys` file. However, in six traces out of 62, AutoPico only dropped the two log files (because the samples was unable to execute correctly), and in four traces it created more than 15 times the same log files in the same location (possibly due to the fact that the sample modified more registry keys, each time recreating the log file from scratch). The ordered list of the number of file creation events for all executions in our dataset looks like the following:

$$[2, 2, 2, 2, 2, 2, 4, 4, 4, \ldots, 4, 17, 19, 19, 20]$$

The Interquartile Range (IQR) and the median absolute deviation (MAD) are measures of spread that are robust to outliers. Unlike the classic standard deviation, these measures are not affected by measurement values that are either too low or too high. For this reason, IQR and MAD are widely used in other experimental fields [32,33]. In our study, a trace may exhibit an atypical number of actions owing to the malfunctioning of the sample, because of the lack of a required component or because the host was shut down mid-execution. High action counts may also occur when the malware was designed to infect all of the files in a directory and it

encounters a few machines with an unusually large number of files,[1] which results in outliers for the number of file actions. The IQR is the difference between the 75^{th} and 25^{th} percentile values of the action-count distribution. In the AutoPico example, the IQR is 0, as it is the difference of the value in the 47^{th} position (a 4 in our example) and that in the 16^{th} position (again a 4) in the ordered list of 62 values. The MAD is the median of the absolute values of each count's deviation from the median. In the example the MAD is 0 as well, because, after subtracting 4 (the median) from each count, we get a vector where 0 is repeated 52 times and there are only 10 non-zero values, and the median of this vector is 0. In contrast, the standard deviation for the AutoPico traces is 3.67, which inflates the action variability that would be reported. Moreover, the variability would be heavily influenced by the four large outliers (17, 19, 19, 20): without them, the standard deviation of the action counts would drop 0.61, while the IQR and MAD would remain 0. This suggests that robust measures of spread, such as the IQR and MAD, are not likely to lead to conclusions biased by artifacts in the data.

At the same time, the tail of the distribution may also provide meaningful insights, e.g. when it reflects the behavior of targeted malware. We therefore select two additional measures, the 90-10 and 99-1 percentile ranges, because they are analogous to the IQR but are gradually less conservative in discarding the distribution tails. In the AutoPico example, the 90-10 and 99-1 percentile ranges are 0 and 17 (19-2) respectively. In our analysis, we compute the MAD, and the 75–25

[1] An example is the authors' fileserver, which stored so many files at one time that it crashed our backup service.

(IQR), 90–10 and 99– 1 percentile ranges. We report one representative measure when the results are similar, and we discuss when we observe differences among the four measures.

Measuring parameter variability. We measure variability on parameters for each action type separately. We then compute the Jaccard index, which is a popular choice to measure the distance between the parameters observed in two malware executions [34,35], on the parameters observed in different executions of each sample. This way it is possible to identify whether similar parameters are chosen (e.g., create a file with the same filename) or on contrary, the parameters are randomized or very different among executions, therefore, malware detection signatures should not incorporate them. We also perform IQR measurements on the count of unique parameter values,to get a precise picture of what, and how, changes across multiple executions.

3.1.2 Finding Behavioral Invariants

As we introduced in the previous section, actions are a common abstraction to represent units of behavior. On top of that, researchers have proposed many different models to build signatures by expressing patterns over sets of actions. While the literature of models is very rich, ranging from simple ngrams or ordered bags [26] to complex graph-based structures [23], the industry still lacks a common framework for expressing and sharing behavioral models (the role that Yara [36] plays for static signatures).

```
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        CommandLine: '*-noni -ep bypass $*'
    condition: selection
```

Figure 3.1: A sample signature from SIGMA.

To the best of our knowledge, the only available resource that contains a sufficiently-large set of signatures of this kind is provided by SIGMA [37], a project that proposes a language to express patterns for log analysis. As OS audit logs contain information about the interaction of each process with the environment (something equivalent in nature to system calls or the abstract actions in our model) the language used to express SIGMA rules allows analysts to write Yara-like pattern over the runtime events of a sample.

By reviewing previous papers and the SIGMA ruleset, we found that a common building block of all these signatures is the ability to check for the presence of an action and match a portion of the its parameters (typically through a regular expression). For instance, Canali et al. [26] use the action type and the full parameter value to create complex signatures. Similarly Trinius et al. construct a representation of malware behavior that uses the action type and parts of the parameters to create a behavior profile for their malware and Trinius et al. [38] use an exact match on their proposed features. This is also the case for SIGMA rules, as the one reported in Figure 3.1, which matches a process creation action in which the command line parameter matches the specified pattern.

Our goal is not to create signatures nor to evaluate different signature models.

Instead, we want to measure which constant elements exist across multiple executions, with the assumption that any good signature would need to build upon these elements and avoid using information from other transient behaviors.

For this reason, we break down each parameter value in a set of tokens according to classic windows delimiters [39]—such as backslashes for directories and spaces for command-line arguments—and study the evolution of each token both individually and aggregated with other tokens extracted from the execution traces. As an example, we observed that around 70% of the SIGMA rules contain at least one of such token, confirming their role of building blocks for more complex signatures.

3.2 Dataset

The dataset we are using is a collection of 7.6M execution traces that the AV vendor has collected across 5.4M real users during the year of 2018. The data is collected by a component of the AV engine that is responsible for behavior-based detection. This component records high-level behavioral data about the executed programs until they terminate or until the system is able to classify them as either benign or malicious and kill. For the sake of validity, our data only includes programs that terminate normally. Therefore, unlike data collected from sandboxes, our data is not limited to few minutes of execution and because the traces are collected from real users, they do not suffer from the limitations introduced by synthetic analysis environment. Our data does not consist of malware samples that were executed intentionally for data collection, but samples that at the time of collection were

	Mal	PUP	Ben
Num. samples	2424	1621	22443
Num. machines	0.5M	0.9M	4M
Num. executions	1.1M	2M	4.5M

Table 3.1: Dataset summary.

not yet known to be malicious or pup and were able to evade the static malware detection solution installed on the computers. Our data is a reflection of the set of threats with which the behavioral detection components need to combat.

Dataset coverage statistics. Each item in our data consists of a sequence of behavioral actions performed by a sample together with SHA-256 hash of the sample, an anonymous machine identifier and a timestamp. Thanks to the unique SHA-256 hashes of the samples, we were able to query VirusTotal (VT) in the following year (2019) of the data collection and identify the corresponding labels assigned to those samples by various AV engines. While we labeled the samples that were consistently labeled as benign by all AV products, we label samples as malicious or PUP using AVClass [40], a state-of-the-art technique for massive malware labeling. From the VT reports obtained in August 2019, AVClass identified $22,443$ benign, $2,424$ malware and $1,621$ PUP samples, as listed in Table 3.1. We perform our variability analysis on execution flows we were able to label with high confidence and those that were observed in at least 10 machines. This experimentally chosen threshold made it possible to accurately measure variability of the sample sample across different machines. 85% of the samples were executed between 10 and 100 machines,

rest were observed in more than 100. The data was collected from computers from across 133 countries: USA(48%) and China (14%) have the largest fraction of our data points.

In Table 3.2 we show the distribution of the Operating Systems for the machines in our dataset. The vast majority of machines run the Windows 7 build 7601 and the rest run a flavor of Windows 8.1 or 10. 55% of the executions happen less than one week apart, while respectively 12%, 6%, 4% and 3% are executions that were collected after the second, third, fourth and fifth week from the initial recorded execution. On the 11% of the samples' re-execution happens 9 weeks after the first appearance of the malware. As a matter of fact, we measure the time variability for the executions happening during the first 4 weeks after the first appearance, covering over 80% of the executions. For instance, a crypto miner sample first appeared on April 5th and within 7 days we had 47 executions from 35 machines. During the next 7 days we captured 18 executions, 4 of which on new machines. In the 3rd week we record the last 7 executions, none of which from any new machines.

Execution statistics. An execution trace is composed of multiple actions. The actions are heuristically-defined behavior units such as file creation/modification, registry key creation/modification, mutex creation etc. In addition to those common behaviors analyzed largely by the literature, we also have some behavioral actions that were defined by the security vendor such as disable Windows defender, disable updates, change firewall options, keylogging, change IE settings etc.

In Table 3.3 we show the top 8 action types in our dataset which corresponds

	Ratio
Windows 7	56%
Windows 10	35%
Windows 8.1	3.1%
Windows Server	2.6%
Windows XP	2%
Other Windows Versions	1.3%

Table 3.2: OS version distribution.

to 87% of the whole data. On average, per execution trace we identified 150 actions out of which 39 are file creations. In our study, due to space constraints we present the action level variability analysis for a subset of these actions. To this end, we set the following criteria:

1. **Action occurs in any execution in more than 25% of the machines.** To measure a non-zero machine variability of a certain action with IQR, it is necessary to observe it at least 25% of the machine.

2. **More than 1 action appears in the executions.** If the action happens only once in executions, it is not possible to measure its variability (the only possible result could be 0 or 1).

   7 of the actions in Table 3.3 meets this criteria.

Ethical Considerations. As mentioned earlier, we did not distribute or launch any samples on the user machines; instead, all the executions in our data set were

	Ratio of dataset
File Create	26%
Mutex Create	20%
Registry Set	14%
ProcessLoad	8%
PECreation	6%
RegistryKeyCreated	6%
DirectoryCreated	4%
ServiceCreation	2%
Others	14%

Table 3.3: Ratio of action types over all the dataset.

triggered by the users, and the malware and PUPs we report reflect real-world attacks agains those machines. The anti-virus detects and blocks all the malicious samples known at the time and collects execution traces for samples that remain suspicious, in a last-resort effort to discover unknown malware. In consequence, we do not cause any harm to the machines in our study that would not have occurred without the anti-virus product and our data collection. Moreover, future updates to the anti-virus will clean the infected hosts once the malware is discovered, owing to the data collection. The behavioral analysis data was collected from users who opted in sharing their data. Necessary anonymization actions are taken to preserve the privacy of the customers. None of the data fields in the dataset contains any identifiable information.

3.3 Behavior Variability in the Wild

In this section, we analyze the variability we observed in the behavior of malware, PUP, and benign programs when executed on different end-user machines. We measure both the action variability and the parameter variability, as discussed in subsection 3.1.1. We first conduct these measurements across space (the differences among a sample's execution traces on different machines) and time (the differences among its traces in different weeks).

In our data, some executions contain duplicates (i.e., the same action type with the same parameter value). This could happen for example when a sample opens the same file multiple times. As these operations are idempotent with respect to the our behavioral specifications, defined in subsection 3.1.1, we perform deduplication on our data before we apply the variability analysis.

3.3.1 Machine Variability

We start by measuring the variability of executions of the same sample across different machines. Our goal here is to understand whether this phenomenon exists and if it does, on which type of executables it is more prevalent. We only look at executions of a sample that happen max one week apart to identify variability that happen only due to being run on different machines not due to time.

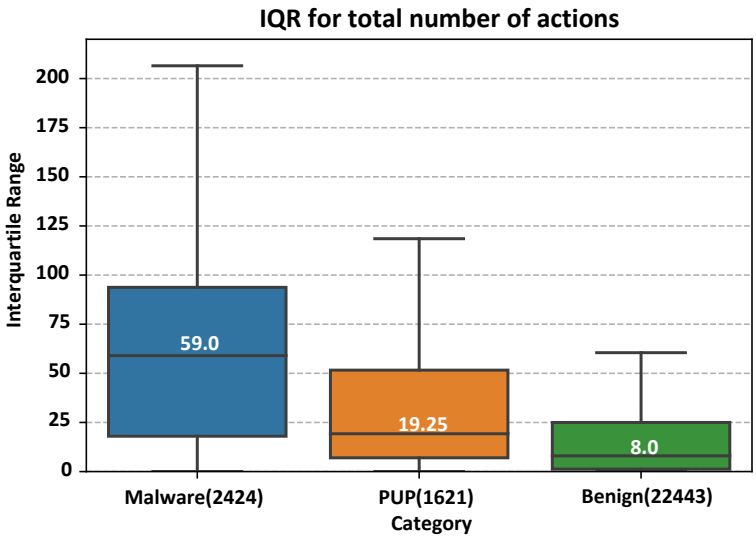

Figure 3.2: IQR action variability for all actions.

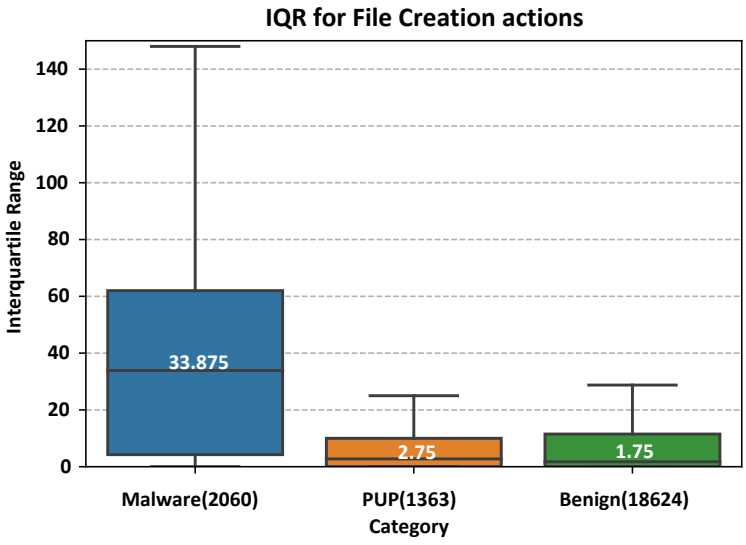

Figure 3.3: IQR action variability for file creation actions.

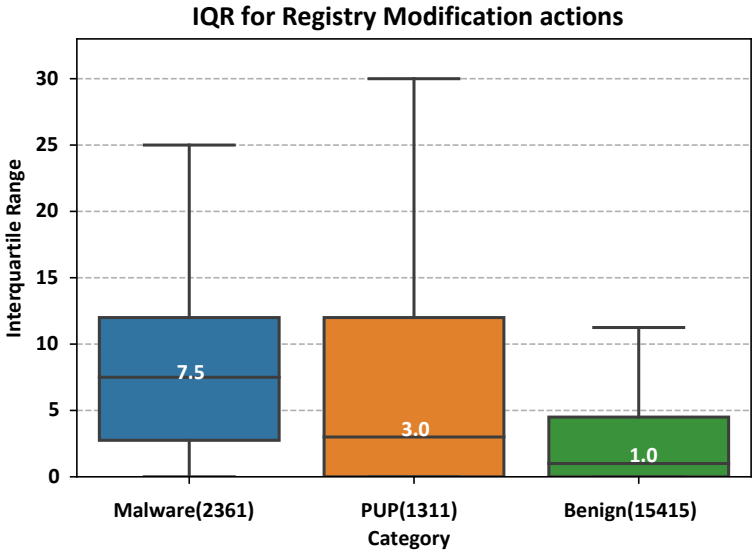

Figure 3.4: IQR action variability for all registry modification actions.

3.3.1.1 Action Variability

We analyze action variability through IQR and MAD. In order to understand the impact of the outliers on the results, we also look at the difference among 90-10 and 99-1 percentiles. The 3 figures above illustrate the distribution of IQR variability, across all actions (subsubsection 3.3.1.1) and only for the two most common actions we observed in our dataset: file creation (Figure 3.3.1.1) and registry modification (Figure 3.3.1.1). The numbers between parentheses for each category is the number of samples that we use for our variability analysis. Because not all samples had file creation or registry key modification actions in their executions, these numbers are lower than the total number of samples. The separate boxplots for malware, PUP and benign samples allow us to compare the action-variability distributions within these categories and to assess the extent of these differences.

To confirm these visual observations we compare these empirical distributions using pairwise U-tests [41], a non-parametric method for inferring whether the samples are likely drawn from distinct distributions. In the paper, we report differences that are statistically significant at $p < 0.001$ level.

Malware exhibits higher variability across machines. We expect to see a higher behavior variability in malware samples, owing to a host targeting, evasion and obfuscation attempts, or the tendency to attempt operations that may fail on some hosts (e.g. privilege escalation). subsubsection 3.3.1.1 confirms this: comparing the three boxes, which represent the bulk of the measurements from each empirical distribution, suggests that the action-variability of malware is typically higher than that of PUPs, which is typically higher than that of benign programs. The median IQR for malware is 59 actions, which means that the top 25% of a sample's execution traces are $>$ 59 actions longer that its bottom 25% traces, for half of the malware samples in our dataset. In contrast, the median IQRs are 19.25 and 8 actions for PUP and benign, respectively. We observe similar trends with the MAD measurements. While the average difference from the median for malware is 35.5 actions, for PUP and benign, it is 10.3 and 2.9 respectively.

This leads to the question *where does this variability come from*? When breaking down the variability according to action types, we observe a striking difference for file creation actions (Figure 3.3.1.1). The median IQR for malware is 15$\times$ larger than for PUP and benign samples, and the bulk of the distribution includes much larger values. In contrast, the variability distributions for PUP and benign samples

do not appear to be different for file-creation actions, while PUPs exhibit more variability for registry-modification actions (Figure 3.3.1.1). This suggests that malware classification solutions based on file creation actions could lead to inaccurate results, as the high variability among the execution traces of a sample may place some of these traces in different clusters; we investigate this in more depth in section 4.8. Conversely, a malware detector able to observe executions on multiple hosts could utilize file-creation variability as an indicator of malicious behavior.

Case study. We refer back to the Ramnit sample in section 3.1 At least 25% of the executions occur on Windows 7 machines where the malware is running with user privileges. Therefore, the malware runs a privilege escalation exploit causing a large number of mutex creations. The rest of the executions happen in different OS version or with admin privileges, thus the executions are shorter. The action variability is affected by the longer executions showing an IQR of 34, which is the number of mutex creations.

> There is a significant variability on the behavior of malware among different machines. When malware detection solutions rely on data collected only from one sample up to 200 (med. 59) behaviors could be underrepresented in the detection model.

Even though malware still shows a significantly higher variability when expressing the variability in terms of the 90–10 and 99–1 percentile ranges instead of the IQR, the action-variability distribution of malware becomes harder to distinguish from the PUP and benign distributions This is not surprising as these measures are not as robust to outliers as the IQR and MAD. The PUP and malware distributions remain distinguishable for file-creation actions, but overlap significantly for registry-

modification and mutex-creation actions. In fact, a few PUP samples seem to have more extreme outliers causing the 99–1 range to be larger than that of malware.

> While malware and PUP both vary more than benign in all the action types, they show variability in different action types.

3.3.1.2 Parameter Variability

We now take a closer look at the variability in the parameter values. We calculate the Jaccard index among the parameters of the same actions types in the execution traces. In this experiment use the full value of the parameters types listed in Table 3.4. Our observation was that the parameters values across different machines have a high variability for both malware, PUP and benign programs. For all of the parameter types, we obtain a Jaccard index is 0, indicating that there is no full value shared across all executions. Note that for this step, we do not normalize the data to remove computer specific artifacts and also do not extract substrings from the parameters. However in section 3.4, when exploring invariants parts among executions of samples, we will perform deeper investigation on the substrings as well.

> No parameter value is shared among all machines except for file extension. An analyst has to rely on substrings/tokens of the parameter values to create signatures.

We also perform IQR measurements, to obtain more insights about the distribution of the variability among different parameter types. In Table 3.4 we report the median and 75th percentile IQRs of different parameter types. We can clearly see that benign programs rarely change the directory they work on, the file names created or the executables they launch. On the other hand, malicious samples tend

		Median			75^{th} percentile		
		Mal	**PUP**	**Ben**	**Mal**	**PUP**	**Ben**
File	Path	4	1	-	10	3	2
	Name	25	2	1	49	8	8
	Ext.	3	1	-	5	2	1
PE	Path	1	-	-	1	1	-
	Name	1	-	-	3	2	1
	Ext.	1	-	-	1	1	-
RM	Key Path	2	1	-	3	3	1
	Key Name	5	2	-	9	6	2
	Value	5	2	1	10	6	3
D	Path	1	-	-	1	1	-
RC	Path	2	1	-	3	3	1
M	Name	6	3	1	9	7	4
P	CMD line	4	-	-	6	1	1

Table 3.4: Parameter(s) IQR variability for malware, PUP and benign. [**PE**] PE File Creation actions, [**D**] Directory Creation, [**RM**] Registry Key Modification, [**RC**] Registry Key Creation, [**M**] Mutex Creation, [**P**] Process Creation actions.

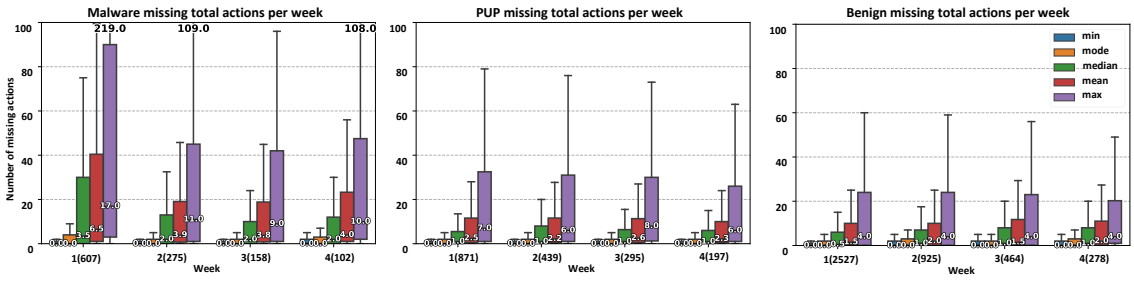

Figure 3.5: Missing actions compared to the previous week.

to create a significant amount of new files (25 new files for 50% of the malware), to work on several directories and even to create different executables over different executions. This finding indicates that the same sample's behavior can vary to an extent that it could be hard to identify a behavioral indicator that is common among all. We will explore this aspect in more detail in the following section.

Case study. In the executions of a Glupteba malware sample we observed that the Jaccard index across multiple machines is 0 for file names and 0.2 for mutex names, while the IQR for file and mutex creation is 0 and 2 respectively. This means that the malware changes significantly the name of files but not their absolute number, while mutex names are more similar but with a larger variability in terms of number. In this particular case, mutexes were a better candidate for building signatures, as we found that `h48yorbq6rm87zot` appeared in all the machines, which is also confirmed by a report from TrendMicro [42]. On the contrary, the mutex `ZonesCacheCounterMutex` and `ZoneAttributeCacheCounterMutex` only appeared in half of the machines, which explains the IQR of 2.

3.3.2 Time Variability

In this section we look at how variability is impacted by the time in which a sample is executed. We start again by looking at the volume of actions and then zoom into those actions by including their parameters into the analysis.

3.3.2.1 Total Action Variability

We measure the action variability by comparing executions of the samples in different weeks. Since 80% of the samples in our data were executed at most four weeks after the first week of their appearance, we perform the per-week time analysis on those next 4 weeks. To simulate what an analyst would deal with, we consider the first week's executions as the base and compare it with each of the 4 consecutive weeks. Here, we cannot use IQR as for each sample we only have 4 data points. We simply count the number of missing and new actions observed in each sample's executions compared to the previous week. The results are reported in Figure 3.5.

Malware have the highest number of missing and additional actions. The general takeaway for coarse-grained time variability analysis is that there is a significantly larger time variability in malware compared to PUP and benign samples.

According to the Figure 3.5, on average across all machines we see 6 missing actions 1 week after the first execution. Even though this number might seem low, depending on what those actions types are variability over time might have an impact on the malware detection solutions. Note that there are also some malware samples that show a tremendous variability (average of max being 17, max of max

being 219). One possible explanation for this significant number of missing actions is that when malware is re-executed on the same machines, might not need to repeat some of the behavior such as creating particular files. At time of the data collection, the malware samples were not yet known and therefore, the machines were not cleaned up before the re-execution. However, we also observe variability over time when looking on the new actions that appear on the following weeks. Similarly, malware samples have on average 1 new action appearing every week, which is larger than PUP and benign. We also highlight that the machine with the maximum number of additional actions seems to have a maximum of 63 new actions and more than 3 new actions for 50% of the malware samples. For malware execution longer than 1 week from their first appearance less new actions appear, indicating a more stable behavior by time.

Case studies. A TOR-connected coinminer was dropping *miners.ini*, *miners.ini.** and *minergate.log* and launching *minergate-cli.exe* before January 18th in 2018. In some machines it was also dropping up to 14 **.tmp* files. After 18th this behavior completely stopped, resulting a number of missing actions in the following weeks. On the other side of the scale, we also identified a Remote Administration Tool, which initially dropped 5 dlls files and an executable (*setacl.exe*) before March 16th 2018. On April 3rd, it started dropping 7 more dll files and 3 new executables. We also have examples for malware that misses and adds new actions at the same. In it's first week of appearance the software was dropping various files on different machines such as *microsoft office*, *foxit pdf editor*, *autocad 2015 qqlivedownloader.exe*. This behavior could be due to the user's interaction with the malware or simply the

malware hiding its purpose. The next week's executions no longer drop any of these files, but *zny_znykb030.exe* or *kuaizip_setup_2523474329_rytx2_001.exe* appears to download consistently in almost all the machines. We believe that in this case the user had no control of the malware and the downloads happened silently. This malware was still running 4 weeks later performing the same actions. A final example is a sample of *Adware.Chinad*, which dropped various files (*microsoft office*, *foxit pdf editor*, *autocad 2015*). On the following week, a new executable (*zny_znykb030.exe*) followed by potentially pirated other software are downloaded.

In malware, time variability is the largest. While the variability is mainly due to the missing actions, there are also new events that appear on the following weeks.

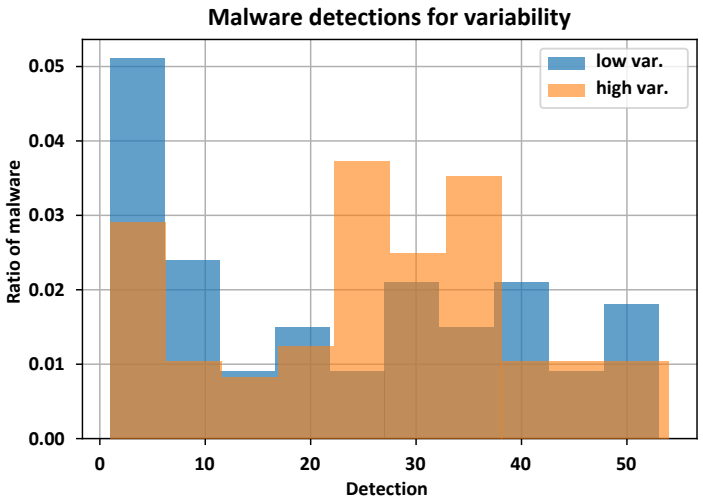

Figure 3.6: **Relation between detection and the time variability.** This result shows that malware that have high time variability have higher VT detections.

Malware with highest detection rates vary more over time. One interesting observation on the malware that exhibit more variability over time was that

in general more of the AV engines would label them as malicious. In Figure 3.6 we show the distributions of the malicious samples with low time variability (lower than 25^{th} percentile) against the ones with high variability (on the top of the 75^{th} percentile). For each of samples, we check the total number of detections in VirusTotal in November 2019 (i.e., one year after the first time we observed the samples). As it can be seen seen, the samples that are AV software eventually detected the most had a higher variability across time. This shows that even if in general time variability is low across the board, for easier to classify samples it seems like time has a more significant effect on the variability.

To get a better understanding of this phenomenon we conducted two case-studies. In the 75^{th} percentile we found a version of kuaizip and analyzed its behavioral data manually. This malware seemed to stop working sometime around the second week of April 2018, after which it still performed host-related actions but failed to download the PE files it was retrieving before. At the other side of the spectrum, we chose a malicious sample that exhibits low variability. We found an open source DLL injection tool classified as malware which performs exactly the same actions every time it runs. This likely-to-be-malicious sample injects into *robloxplayerbeta.exe*, creates *settings.xml*, and sets some registry keys. Upon further analysis we found that this is being intentionally used for cheating in games and this exact behavior is observed over and over again. While preliminary, these experiments seem to confirm that time variability affects the most those samples that rely on an external infrastructure.

3.3.2.2 Parameter Variability

When switching to the fine-grained analysis of each parameter, we now observe a very different picture from the results we obtained by looking at the variability among hosts. In fact, the Jaccard Indexes show that for goodware and PUP there are a large number of perfect matches over time when the sample is re-executed.

Over time, malware actions parameters vary a lot, while PUP' and benign's ones do not vary at all. The difference is remarkable. Even at the median, the parameters of actions performed by benign software change very little, and the 75th percentile they change almost nothing at all. If we consider that the same indexes were zero when considering executions across different hosts, this result emphasize a very important distinction. Goodware creates different files, mutexes and registry keys in different machines. But when we consider two executions on the same machine, those values remain constant. The same phenomenon does not happens for malware, where the Jaccard index is zero across the board, both in case of different machines and in case of different executions on the same host. The only few exceptions to this rule are regarding file paths and file extensions, which still have a low similarity.

If we look at the 75th percentile things get more stable and both malware and benign files show a high similarity. This means that for at least 25% of the samples in our dataset we observe a stable set of parameters at different points in time.

At least 50% of the malware do not reuse same file names, registry keys values and paths, directories in their reexecutions, and 25% execute at least 1 new command.

3.4 Invariant Analysis

Our variability analysis confirms that malware behavior changes over time and on different machines. This indicates that if a behavioral malware detection system is designed with data collected at a fixed time or from a single computer with a particular configuration setting, the real behavior that is common to all possible executions might not be identified correctly. However, as we showed in subsubsection 3.3.2.1, the fact that malware samples carry large variability across different executions does not rule out the possibility of building accurate detection models from behavior that remains stable. Therefore, in this section we focus on measuring the invariant part of malware behavior, to better understand how effective behavioral-based detection systems can be if their models are built upon the right set of events.

Roughly 80% of the SIGMA rules are created from values extracted from file and process creation events, and these two are also in top 7 most popular actions in our dataset. Therefore, due to space limitations, in this section we focus on those actions and their parameters. We identify the invariant behaviors only from the malicious samples as our goal is to evaluate behavioral malware detection techniques. We only use benign samples when simulating a signature generation process, by

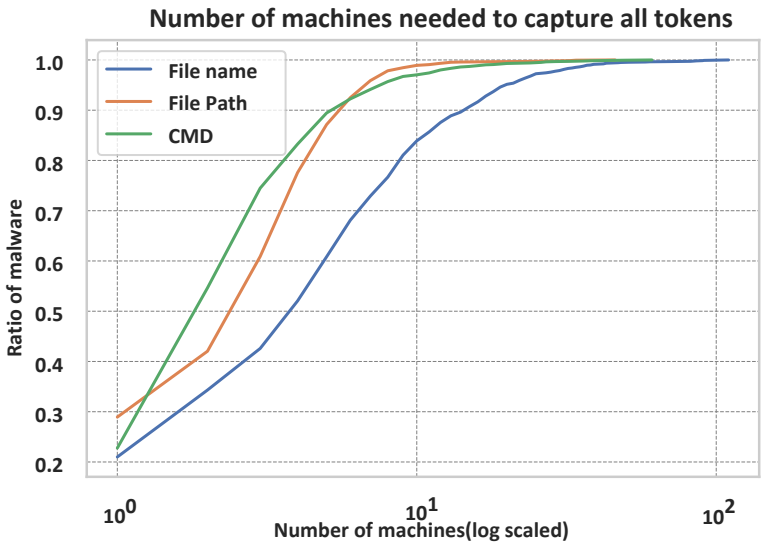

Figure 3.7: **CDF of number of machines and the amount of malware values.** It takes more machines to capture all the CMD tokens than other parameters' tokens.

extracting the invariant parts that are not observed in the benign execution traces.

Beyond full parameter value. In subsubsection 3.3.1.2 and subsubsection 3.3.2.2 we utilize the full value of the parameters to measure the jaccard index. In this section we follow a simple approach to split those values into smaller tokens, explained in subsection 3.1.2, with the aim of finding a shared value across machines. To motivate our approach, in subsection 3.6.7, we show an analysis of the prevalence of parameter full values and their tokens that goes beyond simple Jaccard index.

3.4.1 How Many Hosts Are Enough?

One of the main consequences of the findings discussed in section 3.3 is that for building more effective and accurate signatures it is necessary to collect multiple data points rather than looking at a single trace collected from one environment. While

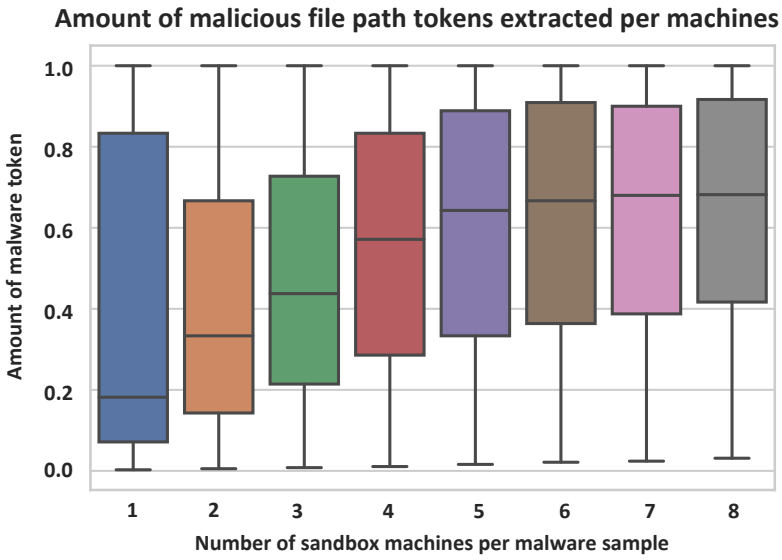

Figure 3.8: **CDF of number of machines and the amount of malware values.** After 4-5 machines we start to get diminishing returns in the number of new malware file path tokens discovered.

this sounds intuitive, to our knowledge, there is no existing study that attempted to measure how many executions from different machines are needed to identify tokens that maximize the coverage of the generated signatures. To this end, we measure the number of executions of the same malware in the wild that can be detected by using the tokens extracted from a small set of executions, as well as the number of those executions that are needed to obtain all the malicious tokens. Based on that, we estimate how many machines are needed to achieve a high coverage of observed behaviors.

Figure 3.7 shows the empirical cumulative density function (CDF) of the fraction of malware samples for which we capture all tokens, for an increasing number of machines (x-axis). We only consider tokens that never appear on benign traces

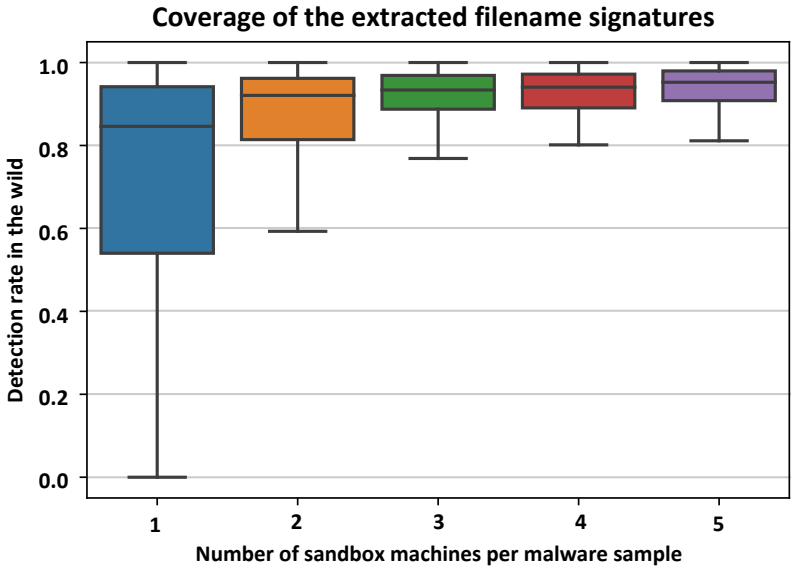

Figure 3.9: **Detection coverage of tokens obtained by combining multiple execution traces for file names or extensions.** The detection rate/coverage of file names or extensions reaches the maximum after 3 to 4 machines.

and we also exclude the unique tokens, i.e., those that only appear in one machine in the wild (as they could be the result of random values). Finally, we construct the CDF by adding first the traces that contain the highest number of tokens, and thus represent a conservative estimate. For 20% of the malware, we need only 1 machine to capture all the malicious command line tokens. If we increase the threshold to 10, we can identify all the command line tokens for 85% of the malware and all the file path tokens for 98% of the malware.

Naturally, the more machines we use the more tokens we can extract, but adding more machines provides diminishing returns. As we can see, for 21% of the malware we can capture all filenames in a single execution, and for 29% of them one execution is sufficient to discover all file path tokens. However, we need to collect

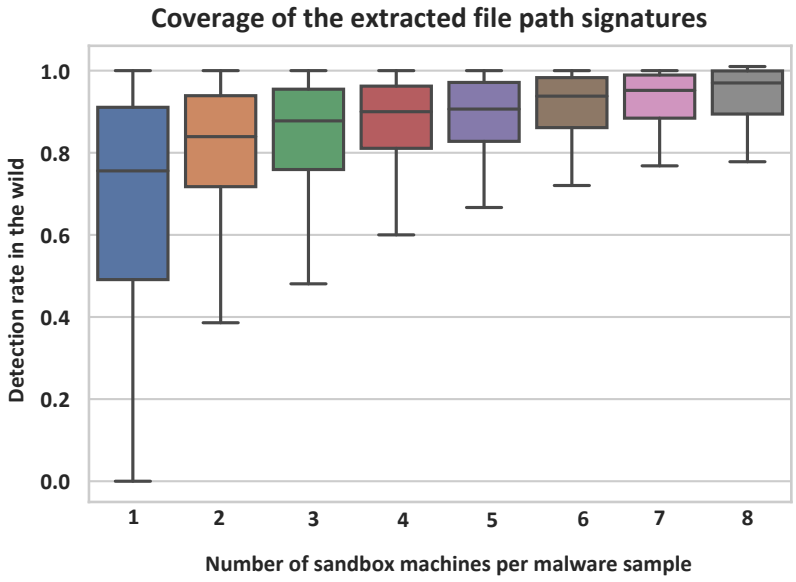

Figure 3.10: **Detection coverage of tokens obtained by combining multiple execution traces for file path.** We need about 7 machines to capture all the malware tokens.

39 traces to observe all the malicious filename tokens that appear in 90% of the malware, while this decreases to only 11 and 17 machines for capturing file path and command line tokens respectively. Since the file path seems to converge faster, in Figure 3.8 we check the impact of the first 8 machines in the amount of tokens we are extracting. We observe that after 7 machines the return of investment becomes small, as for the top 50% of the malware samples we already extracted 68% of the tokens.

While counting the new tokens can give an idea of how many traces we need to compensate for the diversity of behaviors, it does not tell us whether those tokens are sufficient or not to detect malware in the wild. Therefore, we conducted a second experiment. Here we use the tokens extracted from one execution to match

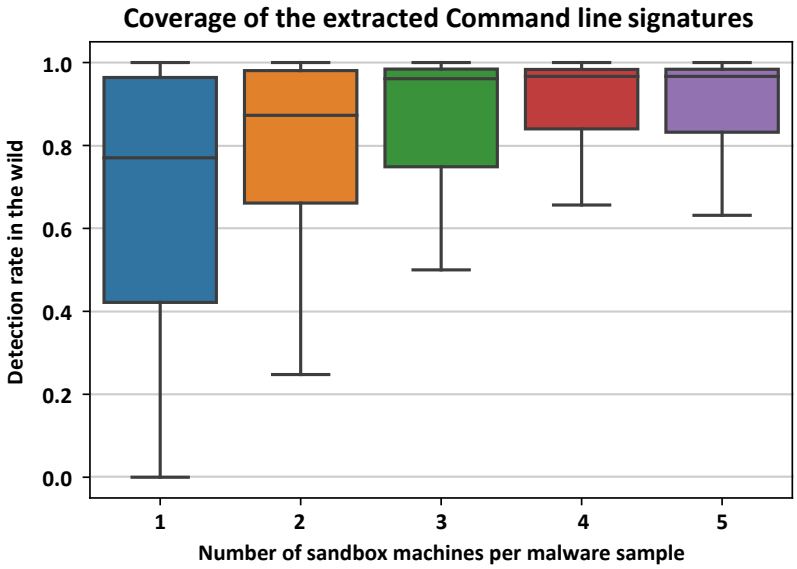

Figure 3.11: **Detection coverage of tokens obtained by combining multiple execution traces for command line.** The detection rate/coverage of command line tokens reaches the maximum after 3 to 4 machines.

the malware traces collected in other machines. If the combination of the tokens can cover all the other executions of the same sample, then we conclude that one execution is sufficient (in theory) to extract a perfect signature. If instead the extracted tokens cannot achieve 100% coverage, we add a second trace collected on a different (randomly chosen) machine in the same week (as for the moment we want to study the machine impacts and not the time impact) and we re-iterate the process. Since the result is dependent on the select machines, we repeat the experiments ten times and report the average.

From the boxplot in Figure 3.9 we can see that while for some malware one execution might be enough, in average the filenames extracted from one trace cover 82% of the executions and the value decreases to 77% if we use path information.

However, the execution traces collected on three different machines are sufficient to achieve the highest coverage when using file name as the parameter. Similarly we find that it takes four machines to saturate the coverage for the command line and seven for the file path. The respective results can be found in Figure 3.10 and Figure 3.11.

> Our results suggest that an analyst should analyze the malware in 3 random virtual machines to capture most of the file names, 4 for CMD line and 7 for file path. A possible way to generate such random machines, instead using the same machines for all malware, may be to use a random vm generator like SecGen [43] with the features proposed by Miramirkhani et al. [44].

3.4.2 How Soon Should We Re-Execute?

We now investigate the re-execution interval needed to achieve the best coverage in the wild. This is more difficult to measure, as it represents a trade-off. If you re-execute the sample too early, you may learn little and your signature may not catch the behavior that the malware will exhibit in the future. But if you re-execute the sample too far in the future, than your initial model might get outdated before you re-analyze the sample.

For this analysis, we take a first execution trace during the first week of appearance of the malware. Then we collect a second trace on the same machine, varying the time between one and four weeks in the future. We then use the tokens extracted from the first execution to match all malware executions until we

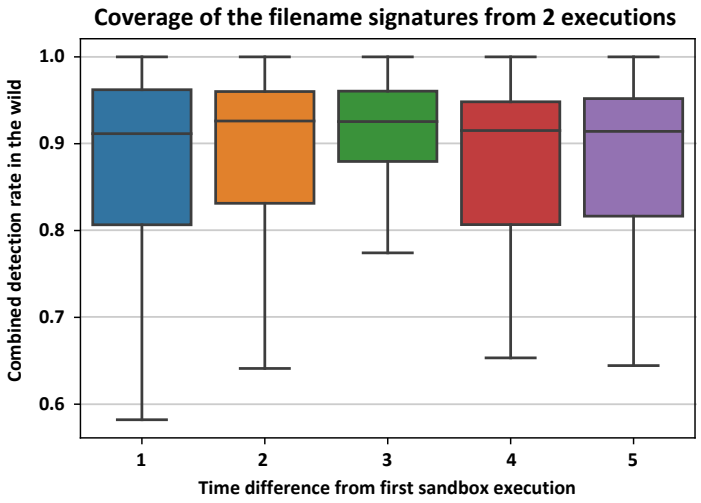

Figure 3.12: **Total filename signature coverage for re-execution intervals.** A periodic execution of every 3 weeks yields the highest coverage across all malware.

re-execute the sample. From that time on, we incorporate the information of the second execution and update the signature to be used for future executions.

Figure 3.12 shows the results for the filename tokens. While the median detection does not change much, re-executing after three weeks provide more benefits (the minimum detection and the 25 percentile are much higher, which suggests that for some malware this makes a big difference).

For the file path the difference in the re-execution interval is smaller, which means that we need more machines to get better detection. However, even in this case we still notice a slightly smaller range when re-executing on the 4^{th} week, which means some malware show a different behavior around that time. The results are the same for command line arguments, where in week 4 we have a more impactful increase in detection, suggesting that malware will be spawning new processes or using different parameters one month from their first appearance.

An analyst should re-execute a sample between 3–4 weeks apart. However, having multiple executions in different days provides less useful information about the malware behavior than having different executions on different machines.

3.4.3 Hunting for the Most Invariant Artifacts

As we showed in previous sections the number of file creations is not a good metric to profile a malware sample due to variability. Similarly, the same file name doesn't appear in all machines. Using more than 1 file name to profile malware seems like the right choice. While using both variant and invariant features is not going to affect the performance of the detector, we need intuition to be sure we have an invariant in our signature. In this section, we measure the covered machines that individual tokens can detect the malware on. For this we extract the malware tokens in the first week and compare their performance to executions happening in the following weeks, to simulate the scenario where an analyst creates a signature with 1 token and deploys it. We show the average effectiveness of each token. We don't remove the random tokens to show the amount of randomness that an analyst has to deal with for each parameter.

We measure the file name token coverage for file writes. The results show that most of the tokens are random and having more than 1 machine allows the analyst to remove them. We noticed that the tokens with the highest coverage were the extensions, therefore we encourage the analysts to split the file name using the dot(.) delimiter and remove the known benign extensions to obtain highly

performing malware file extension signatures. Random tokens happen more often in file names than any other parameter, which means that an analyst should have more than 1 execution to remove the tokens that appear only once.

We noticed that malware tends to write to non-random and non-benign paths. However, there is no clear trend to which subdirectories and on which level are invariant to the malware, therefore, an analyst will need multiple values to construct a signature based on the file path. While we couldn't identify a heuristic to pick the better path tokens we noticed that on average, for all malware, 25% of non-benign subdirectory names (tokens) appear in all the machines. This means that an analyst will achieve a better detection using a subdirectory name to detect malicious file write than the file name, extension or even command line of process creation. Our study reveals a source for constructing high-performance detection rules using file extension tokens, which future generations of malware may no longer posess. We also reveal the success of file path tokens in constructing a malware detection signature.

3.5 Discussion and Limitations

Impact on State-of-the-Art Solutions. In our paper, we performed an extensive analysis about behavioral variability on malware, concluding that to observe the complete behavior of malware it is necessary to run the malware on several machines repeatedly over time. We conduct two further experiments to illustrate the impact of our results on state-of-the-art solutions.

First we reproduce the experiments conducted in one of the most cited behavioral malware clustering techniques [45]. Such clustering techniques commonly rely on only one execution trace per sample. Note that our goal here is not to call into question the results of the prior work, but to demonstrate the effects of variability in a typical malware-clustering experiment. When evaluating this technique on our data, which consists of several executions of the same malware samples, we observe that one third of the samples exhibit sufficient variability in behavior that their traces were scattered among multiple clusters, thus decreasing the accuracy of mapping samples to the correct family. This suggests that we must be cautious when drawing conclusions from clustering experiments, as the results may be inaccurate if the experiment utilizes a single trace per sample. The details of this experiment can be found in subsection 3.6.5.

In a second experiment, we assess the impact of behavioral variability on the accuracy of anomaly-detection approaches. In this case, we selected Access-Miner [46], a popular solution for building models based on benign execution alone. It is interesting to note that although variability was not explicitly discussed by the authors, Accessminer correctly accounted for it by including multiple execution of benign software collected from different real-world machine.

Again, we repeated the experiments by following the technique explained in the paper (the details can be found in subsection 3.6.6), training the AccessMiner model with a progressively increasing number of traces from benign files in our dataset. Our results suggest that behavioral variability of benign programs also impact the detection rate and that only few executions are insufficient to build and

accurate model. Moreover, our experiment shows that to improve the accuracy of the models and reduce false alarms, it is necessary to also include lower-reputation and low-prevalence benign files to the dataset. In the original AccessMiner paper, only traces from popular benign files behavior were incorporated into the anomaly detector.

Alternative Solutions to Account for Behavioral Variability. Our findings suggest that the more accurate way to collect information about malware behavior is to record program executions on real end-user machines. However, the main drawback of this solution is that known malware needs to be blocked to guarantee the user security, and thus the data collection is limited to files that other methods cannot classify one way or another.

Other options exist for researchers to account for the behavioral variability of malware. For instance, *Multipath Exploration*, proposed by Moser et al. [47], allows to automatically explore multiple execution paths of the malware binary in the same system. As this method is capable of triggering hidden functionalities, it can replace the need to observe the behavior over different machines and at different points in time. However, this solution is complex and has a very high performance overhead, which makes it unsuitable for large-scale experiments.

Similarly, *Symbolic execution* could be used to trigger unobserved behaviors during malware analysis, such as in the case of time-triggered malware [48]. While this can also help an analyst to tackle the issue of behavior variability, similarly to the multipath exploration solution, symbolic execution is difficult to scale due to

the large overhead and the state explosion problems [49].

A more practical solution consists in running the samples on *different VMs*, with different configurations. While still resource intensive, this method has comparably lower overhead than the previous approaches, making it is easier to apply to a large number of samples. As we showed in section 3.4, we suggest running the malware in at least three (and for better coverage even seven) different VMs to capture significant machine-induced variability. We also suggest the analyst to re-execute the samples at least every three weeks to capture any time-induced variability.

Threats to validity and limitations. Our study carries some limitations due to the nature of the data that was provided by the security vendor. The data was collected from users who have installed the AV product, who might be in general more careful with the security of their computers and, therefore, might be less exposed to attacks. Although we cannot rule out the possibility of selection bias, the large size of the population in our study, the large fraction of malware (9.15% of the unknown samples that could not be classified with other means), and the large spectrum of variability that we observed in the experiments, suggest that our results have a broad applicability.

Our data consists of executions of Windows PE files and therefore, our findings might not apply to the behavior of programs that run on other platforms (Linux [50], Android [51], IoT, etc.). Another unfortunate limitation is that the data does not contain network events. Previous work [21], however, has already shown the existence of a high variability in the network events and discussed its impact on the

overall behavior of malware. Since our goal is not to establish a root cause for the behavior variability, the lack of network data does not impact our main findings. We expect to actually observe higher variability on network events.

All samples in our dataset were not flagged neither as benign nor malicious at the time of their collection. Therefore, the data does not include popular benign programs and malware that can be detected by traditional means (i.e., AV Engines). While this might be seen as a limitation because easier to label programs might not show similar variability on their behavior, the set of samples we analyzed also make our study more unique in its nature. We only analyze those programs that need to be detected by looking at the behavior. In reality, samples that can be identified simply by other means, such as static signatures, do not require a behavioral analysis in the first place. Even if our measurement does not capture the variability of those samples, the impact on behavioral detection would have been irrelevant. Moreover, since our goal is to study variations in the runtime behavior, the analysis can only be performed if a sample is executed multiple times, both in the same environment and across a different set of machines. Therefore, polymorphic samples (in which each SHA-256 hash is only observed once) cannot be included in our analysis.

A recent work [40] has shown that for the vast majority of malware samples its impossible to identify a family name, owing to the use of generic signatures and to inconsistencies among the AV labels. Our clustering experiment provides further insight into this challenge. As we were unable to find a family name for most of the samples in our dataset, we did not study the behavior variability across malware families.

When measuring the time variability, some actions may not re-occur. For example, the malware might not recreate files already created in the previous runs, resulting in a significant number of missing events in following runs. However, our results show that during the re-executions of the same sample we often observe *new* events, thus confirming the existence of variability over time.

Finally, our study might have missed malware that can compromise the kernel of the operating system to evade the AV data collection component. This is common to all studies performed on AV telemetry, and since we do not have control over the execution environment we cannot verify the extent of this problem.

3.6 Additional analysis

3.6.1 Other Dataset Statistics

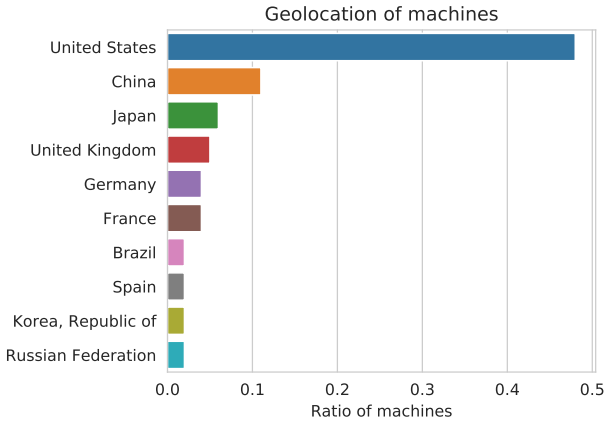

Figure 3.13: Geolocation of all machines in the dataset.

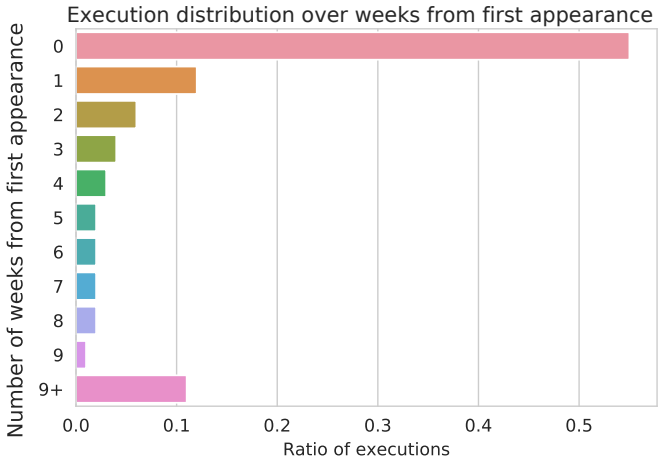

Figure 3.14: **Execution occurrence from the first week (in weeks).** More than 50% of all the executions occur within the first week of a samples' appearance.

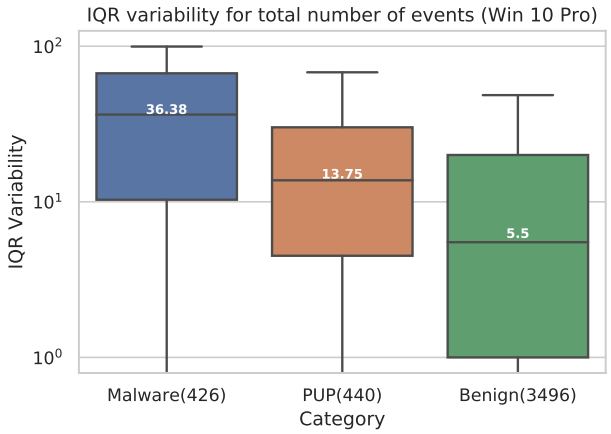

Figure 3.15: Count of all action on Win 10 Pro

3.6.2 Machine Variability for the Top 3 Most Common OS Versions

Unlike prior work [35], which has shown that malware behavior varies due to environment factors (ie. installed Java, .NET, IE6 etc.), we cannot control the sample execution environment. The real-world nature of our data does not allow us

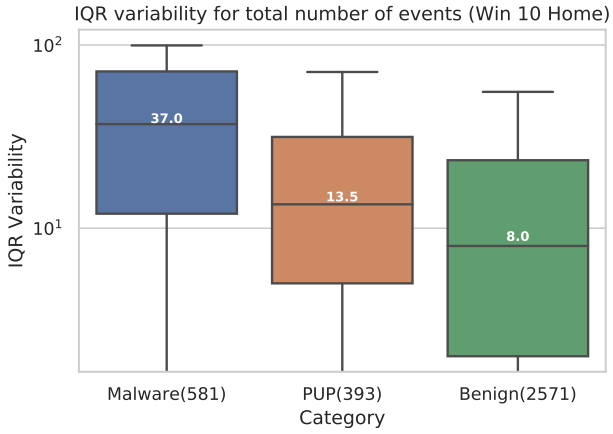

Figure 3.16: Count of all action on Win 10 Home

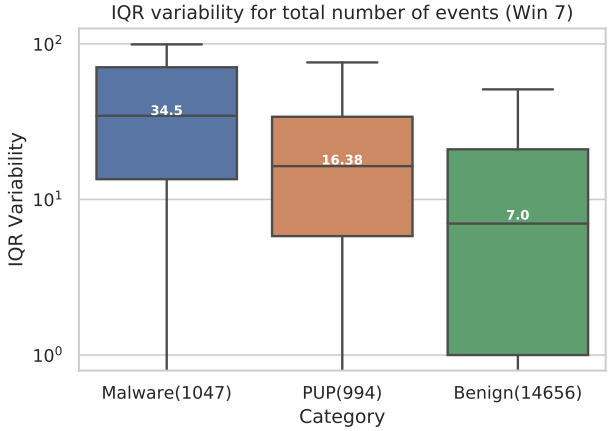

Figure 3.17: Count of all action on Win 7

to measure variability of a sample across different OS versions as it is impossible to understand whether the variability is due to the machine itself or the OS version it uses. Instead, we investigated the machine variability for samples running in certain OS versions and did not find a significant difference. We could not identify a significant difference in the variability distributions of different OS version for any of the action types. This suggests that there is no OS version for which the malware have less variability. Note that we lack knowledge of executions that completely

fail to start. Therefore, we cannot capture executions of samples that did not run on particular OS versions. No matter the OS version of the sandbox the analyst uses, the machine variability is the same for all malware where the sample doesn't fail to execute. Therefore, the analyst will have to account for machine variability regardless of the OS version of their target platform.

3.6.3 Coverage of other parameters' signatures

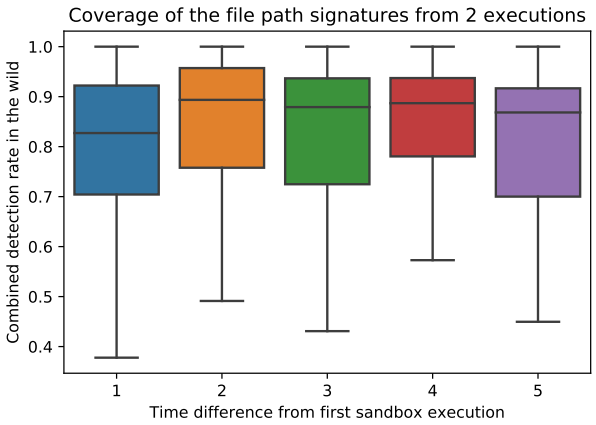

Figure 3.18: File path signature coverage for multiple re-execution intervals.

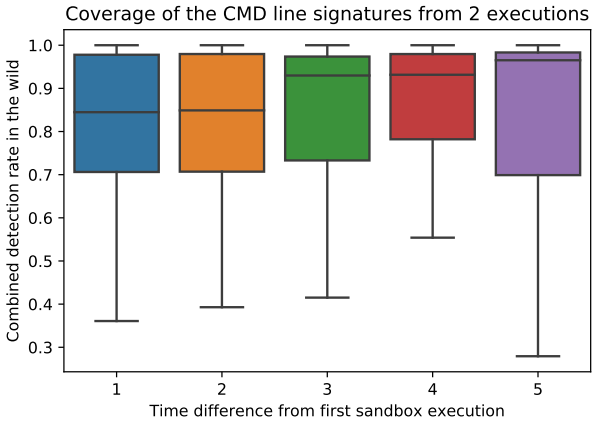

Figure 3.19: Command line signature coverage for multiple re-execution intervals.

3.6.4 Number of Tokens in a Trace

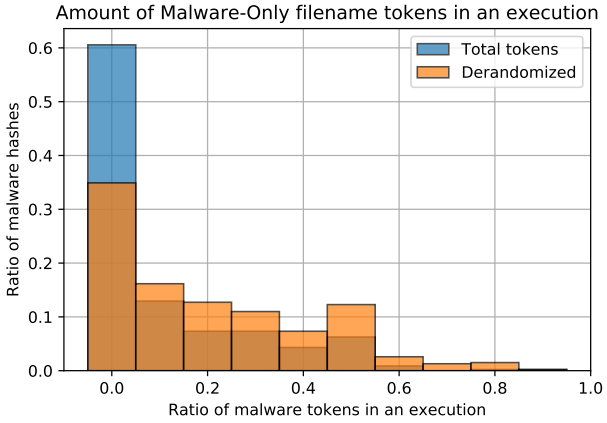

Figure 3.20: File name tokens in a malware trace.

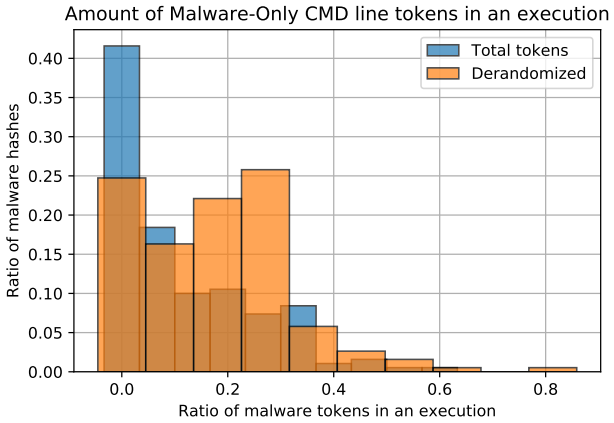

Figure 3.21: Command line tokens in a malware trace.

3.6.5 Implications of Variability on Malware Clustering

Dynamic malware clustering [31, 45, 52, 53] aims to identify malware families (or variations withing the same family) by grouping together samples with similar behaviors. These approaches commonly rely on only one execution trace per sample. Therefore, we investigate how the large variability among the traces of each sample could influence the results reported from clustering experiments.

This can be performed by clustering execution traces, and then verifying whether the traces of the same sample are clustered together or they are scattered among multiple clusters. For this experiment, we implemented the clustering technique described by Bailey et al. [45], which also uses similar features to our dataset. As suggested in the paper, we apply their normalized compression distance to our samples, and we utilize the same hierarchical clustering algorithm and the same method to determine the number of clusters. We clustered execution traces from 2,424 malware samples. For each sample we randomly select 4 traces collected

in the same week but on different machines; we repeat this step 10 times. We cluster the resulting 9,696 traces, and we obtain 88–105 clusters, of which we pick the median with 93 clusters. To interpret these clusters as families of malware samples with similar behaviors, it is necessary that all executions of a sample fall within its family cluster. In average we found that for 67% of malware samples all 4 executions appeared indeed in the same cluster. However, one third of the samples exhibit sufficient variability in behavior that their traces appear in multiple clusters: 27% fall into 2 clusters, 5% in 3 clusters, and 1% in 4 different clusters. This calls into question the conclusion that the behavior clusters reflect malware families. Because some samples exhibit too much behavior variability to be clustered correctly into families, we must be cautious when drawing conclusions from clustering experiments. Importantly, this threat to validity comes to light when we cluster multiple traces per sample, but remains hidden when using only a single trace per sample.

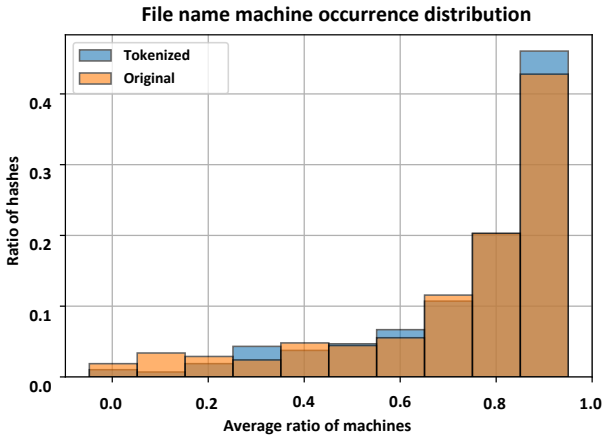

Figure 3.22: The average ratio of machines the parameter values appear on for file names.

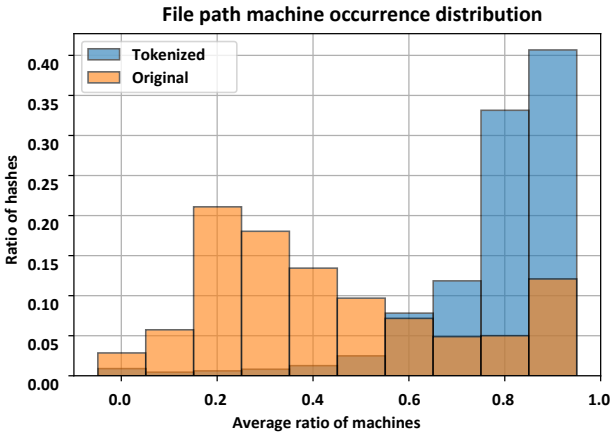

Figure 3.23: The average ratio of machines the parameter values appear on for file paths.

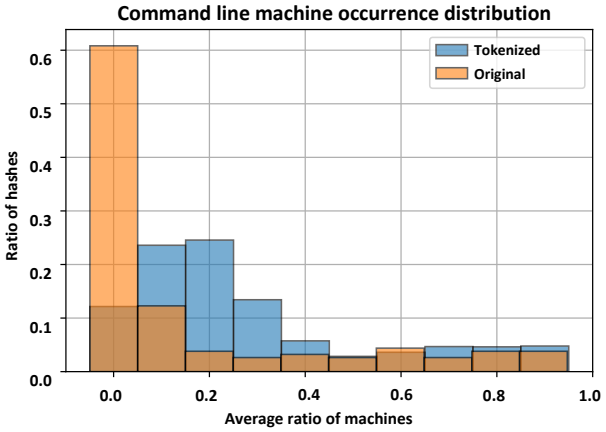

Figure 3.24: The average ratio of machines the parameter values appear on for command lines.

3.6.6 Impact on Anomaly Detection

One way of detecting malware regardless of their variability is to detect deviations from benign behavior. In this category, Lanzi et al. proposed AccessMiner [46], as system-level anomaly detector based on behavioral traces of benign programs. It

is interesting to note that the authors already adopted a technique that accounted for behavioral variability over time and different machine profiles. Similarly to our data, their dataset was also collected from real users but their goal was not to study changes in the application behavior but to obtain a complete picture about how benign files interact with the underlying operating system.

Since in the AccessMiner paper the authors did not discuss how many executions of benign programs are needed to train the anomaly detector, we decided to leverage our data to find an answer to this question such that security companies that opt for anomaly detection rather than malware detection could benefit from our results.

Following the AccessMiner approach, we construct the benign profile by using 90% of the benign executions in our dataset. Remaining 10% is used to measure the false-positive rate. As AccessMiner found file write events to be the most successful in identifying malware, we first build the graph using the file write actions in our dataset.

We first start with measuring the success of an anomaly based model that relies on only one execution per benign sample (Figure 3.25), then the success when all of the executions available to us included(Figure 3.26).

As it can be seen from the figures, a single random benign execution is not sufficient to train an anomaly detector, because it treats most of the executions as anomalies.

The detection rates we obtained from this experiment are lower than the ones reported in the original paper. Concerning that the nature of our data is very

different to the benign dataset of AccessMiner this is actually expected. Note that our data consists of unpopular benign applications, whose behavior might be more similar to malicious and unwanted programs. To obtain a better behavioral coverage for benign programs, not only popular benign files such as those used in AccessMiner should be consider but also lower reputation, lower prevalence benign files.

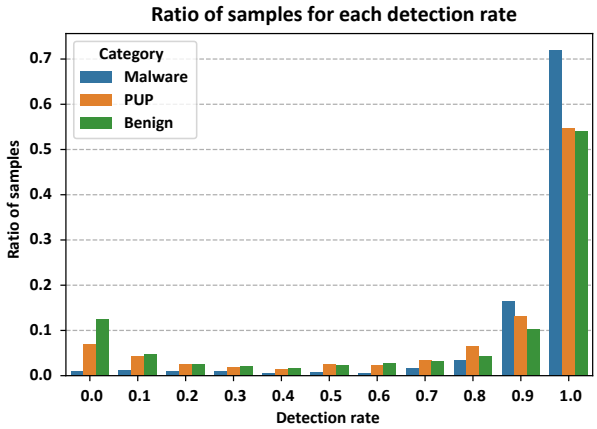

Figure 3.25: Amount of samples for the ratio of machines with anomalous file write (Using 1 random benign execution).

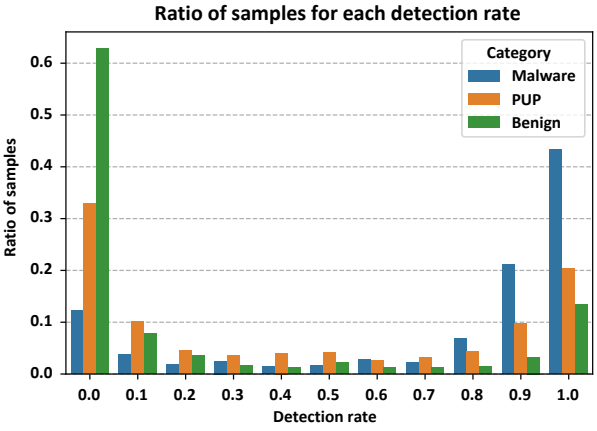

Figure 3.26: Amount of samples for the ratio of machines with anomalous file write (Using all benign execution).

3.6.7 Coverage of Parameter Tokenization

As we explained in subsection 3.1.2, for our signature analysis we break down the action parameters into smaller substrings we call tokens. For instance, file names are divided at the dot character to, while file paths are separated according to the backslash sign, and command line are separated according to the windows delimiters [39].

Our first goal is to investigate to which extent the whole parameter value (in the previous example the entire command-line) remains constant across executions, and to see how the results change after the value is tokenized (i.e., when considering the arguments separately). We plot the average amount of the machines in which each value appears unchanged.

In Figure 3.24 we show the results for the command execution action. In 61% of the malware, executed commands never appear exactly the same way in other machines. Unfortunately, these exact command matches account for 23% of the signatures in SIGMA rules. However, if we break the command in smaller tokens, then we see that in only 10% of the cases these token never appear unchanged. In other words, this confirms the intuition that some parts of the commands are likely to remain constant while others change across machines.

For file creation actions we consider the path and file name separately in Figures 3.22 and 3.23. An interesting finding is that even though the paths in our dataset are already normalized by removing machine specific artifacts like the username, the full path is rarely constant. Instead, file names often remain unchanged

across executions on different machines.

However, after we split the path in subdirectories, we noticed that most of the tokens remain the same (the blue bars on the right side). In fact, at a closer look we found many paths that include random folders in the *temp* directory (e.g.., `C:/Users/username/AppData/Local/Temp/RANDOM`).

> In general, both commands and paths contain some random components that prevent the full value from being observed in different machines. However, if we compare individual tokens we see that a predominant part remain constant, making them good candidates to build signatures.

3.6.8 Mode Execution Spread Across Machines

Introducing mode spread The MAD and the percentile ranges retain the dimensionality of the action counts. Therefore, we also compute the mode spread, a relative measure that allows us to compare the variability of samples that produce traces of different lengths. The mode is the most frequent value in the distribution, and the mode spread indicates how often this value occurs. The mode spread is a conservative and robust measure, as it discards the outliers and all the values that are not equal to the mode. For example, if a malware drops 5 files in 10 executions and a different number of files in 10 other executions the mode spread is 50%. As the mode is the most popular value, and the most likely to be observed by an analyst trying to create a detection signature for a malware sample, the mode spread, intuitively, reflects how representative this value is for the sample's executions in the

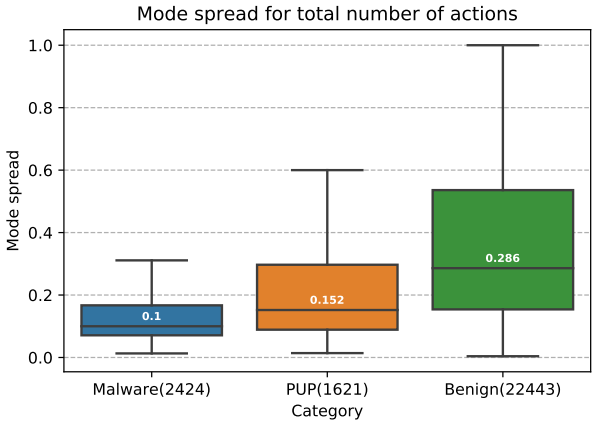

Figure 3.27: Mode spread of the total number of actions in a trace.

wild. In the AutoPico example (subsection 3.1.1), the mode-spread is the frequency of 4's over the length of the list, which is $\frac{52}{62} \approx 0.84$. There is an 84% chance that an analyst seeing 4 file creations in the sandbox will do so in the wild as well.

In Figure 3.27 we compute the mode spread for the total number of actions in an execution. We noticed that the execution with the most popular action count is not very popular across machines for all categories. Worse, malware tend to be less consistent. The takeaway from this is that across machines registry key creations have the highest mode spread in malware. This indicates that an analyst is more likely to find the same number of registry keys being created across machines than other actions. For 50% of the malware samples the most common execution creates same number of registry keys in 58% of the machines.

The results show that if an analyst is to use the number of actions of a the most stable action (registry create) they will be able to capture the malware 25% of the malware found in the wild in 79% of the executions. For the rest of the malware this feature is less promising.

Chapter 4: SCAVY in privilege escalation exploits

In this chaper I will discuss the design, implementation and evaluation of our framework. SCAVY is a framework designed to evaluate the effects of kernel vulnerabilities that allow attackers to corrupt certain kernel structures. This system emulates the memory corruptions and detects the ones that achieve privilege escalation. In the following sections, I discuss the problem statement (section 4.1), background (section 4.3), design (section 4.5), implementation (section 4.6) and evaluation (section 4.7). In section 4.8 I discuss the implications of SCAVY, its limitations and the future of kernel exploitation. At last, I conclude the chapter with a summary of the findings and takeaways conclusion (chapter 5).

4.1 Problem Statement

Despite important advances in *discovering* software vulnerabilities, incorrect assessments of their security implications remain common and can have a pernicious impact. For example, expert recommendations for prioritizing patches [54, 55] initially omitted `CVE-2017-0144`, the vulnerability later exploited by WannaCry and NotPetya; `CVE-2019-2215` was discovered in 2017 but was not patched in many Android devices because its security implications were unknown, before it was exploited

by the NSO group [2]. *Exploitability* assessments are challenging because they require reasoning about state machines with an unknown state space and emergent instruction semantics [56], known as "weird machines". For privilege escalation, in particular, exploitability assessments hinge on the ability to overwrite specific memory targets.

4.1.1 Problem Definition

SCAVY aims to solve a problem of finding memory targets that can lead to privilege escalation. In our context, a memory target is a field of kernel data structure and privilege escalation is defined as a change of a privilege that allows access to an unauthorized resource without using legitimate methods (e.g., permission changing APIs). To facilitate the discussion, we divide a program's execution state into three different states based on how an exploit progresses to achieve its ultimate goal (e.g., taking over the system's control).

1. Before an exploit starts, a process's execution is in an **unexploited state**.
2. After an exploit triggers a vulnerability to corrupt system states to escalate privilege, granting access to resources the exploit wants to compromise, the state becomes an **exploitable state**. This state has access to the exploit's target resources, meaning that the exploit has *achieved all accesses needed* for its ultimate objective but *has yet to achieve* it (i.e., has not compromised the system yet).
3. After the exploit achieves its objective, the execution state becomes an **exploited state**.

Threat Model. We assume a local unprivileged adversary seeking to use a vulnerability in the Linux kernel with a *memory target by SCAVY* to escalate privileges. This threat model is relevant in various settings, such as cloud computing (e.g., Docker), Android [57], and malware exploiting kernel vulnerabilities [58]. As SCAVY finds memory targets for any given exploits, our threat model includes a wide range of diverse memory corruption capabilities of exploits. Note that it also means that SCAVY found memory targets may require vulnerabilities with certain capabilities. As described in subsection 4.2.1, creating an end-to-end functional exploit from the SCAVY identified memory targets requires manual efforts. Automating the exploit generation process is out of the scope.

4.1.2 Unexploited, Exploitable, and Exploited States

Figure 4.1 illustrates the two critical states for our problem definition (i.e., unexploited and exploited states), including privileges of various resources under the states. We first define five different types of privileged resources in Figure 4.1:

- **Read-only Resources**: These are the files that are configured to be read-only (e.g., Apache [59]'s configuration file are read-only to prevent unauthorized modifications [60]). `setuid` files are also read-only.
- **Inaccessible Resources**: Sensitive files or root owned files/directories (e.g., `/etc/shadow` or `/root`) are not readable and writable by an unprivileged process.
- **Privileged System calls**: There are system calls specific privileges (e.g., `mount` and `chroot`). If an unprivileged process calls them, the request will be

denied and failed.

- **Kernel/Other Processes' Memory**: An unprivileged user process is not allowed to read/write kernel memory. Other processes' memory is also not accessible (i.e., cannot read/write) due to the memory space isolation.

We define the three states with privileged resources.

Unexploited State. An execution under this state follows the permission configured for each resource. For example, as shown in Figure 4.1-(a), it cannot read or write inaccessible files/folders (e.g., `/etc/shadow` and `/root`) and cannot write read-only files such as `/etc/passwd`, without calling permission/privilege changing APIs. Kernel memory, privileged system calls, and memory of other processes are inaccessible.

Exploitable State. Suppose a prohibited operation (e.g., read/write) on any of the privileged resources in the unexploited state becomes available *after corrupting a memory target*. In that case, we consider that the execution's state has changed to an *exploitable state*. Note that there exist multiple instances of different exploitable states depending on which resource's privilege is escalated. For example, an exploitable state can have escalated privilege on `/etc/passwd` while another exploitable state provides access to a prohibited memory area. Hence, depending on the exploit's ultimate goal and its environment, a particular exploitable state would be needed. To this end, describing the exact *escalated privilege* and *prerequisite conditions* of an exploitable state is important to determine whether it can be used to achieve the exploit's goal.

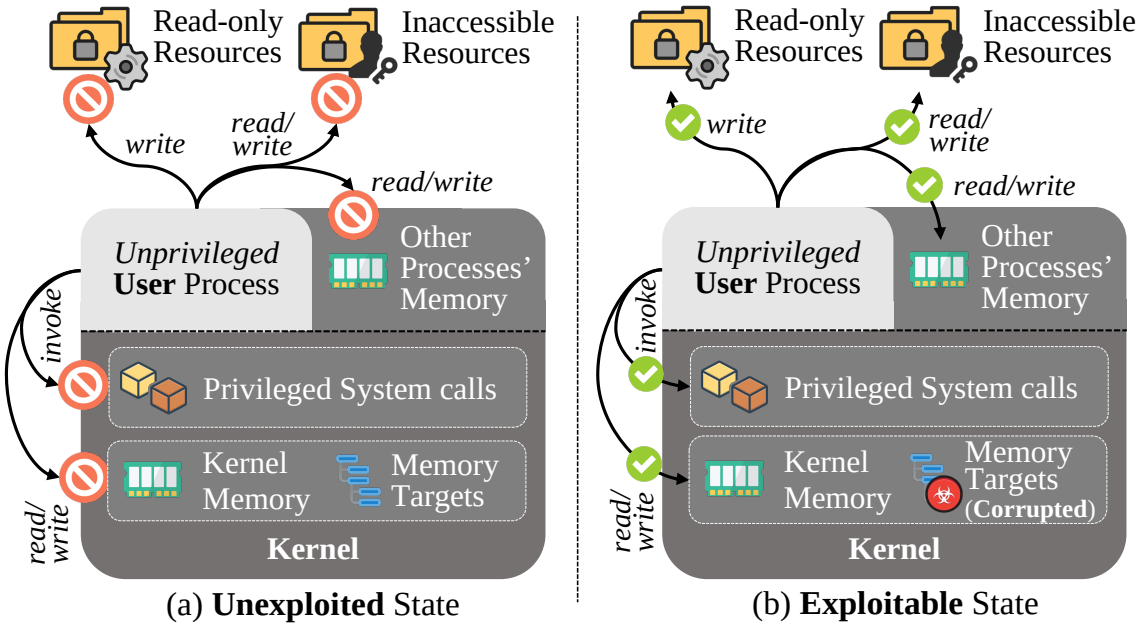

Figure 4.1: Unexploited and Exploited States

Exploited State. An exploitable state becomes the *exploited* state, if the exploit achieves its ultimate goal with the escalated privileges (e.g., adding a root user with the escalated `/etc/password` and `/etc/shadow`). Note that the definition of the exploited state is *specific to each exploit's goal*. For example, to change configurations of Apache, an exploitable state execution with write access to `httpd.conf` is required.

4.1.3 Goal and non-goals

Goal. SCAVY aims to find a memory target that can change an unexploited state to an *exploitable state* when the target is corrupted. To clearly define the memory target's applicability, it is important to identify (1) the prerequisite of corrupting the

memory targets and (2) the post-conditions (or escalated privileges) of the corrupted memory targets. In particular, the prerequisites and post-conditions are defined as follows.

- **Prerequisites**: (1) Required privilege and method for allocating the memory target, (2) Privilege required for corrupting the memory target (e.g., memory write permission if the target is read-only), and (3) Memory corruption capability (i.e., corruptible memory location, size, and possible values) for the memory target.
- **Post-conditions**: (1) Escalated privilege/permission, (2) Name or path of the escalated resource, and (3) Area (i.e., offset and the range) of the escalated resource.

Non-goals. SCAVY does not focus on identifying new vulnerabilities. SCAVY's new memory targets may allow us, in some cases, to repair exploits that have been rendered inoperable by system-level defenses (e.g., SMAP), but they do not improve the exploits' *capability* or *reliability* (as defined in Section 4.4). While we demonstrate the ability to transit from an *exploitable* to an *exploited* state by developing a few functional exploits (see section 4.2), automating this step is out of our scope. Finally, our goal is not to construct an automated *end-to-end* exploitation tool.

4.2 Motivating Examples

In this chapter I will describe two exploits using memory targets identified by SCAVY, to motivate its impact and practicality.

4.2.1 Corrupting `vm_area_struct::vm_file`

Target Kernel Structure. When a user opens a file, the kernel creates memory buffers to hold the content of the file. However, the kernel also provides the ability for users to directly map the contents of a file into their chosen virtual memory pages, making file access as simple as any other memory access. The kernel then ensures that any changes made to the memory will be written back to the mapped file. This is done by using the `mmap` system call with the file descriptor returned by the prior `open` system call. The syscall will create a memory buffer containing the file content, and if it is created with the write permission, changes to the buffer will be written back to the original file, when the file is closed.

The Linux kernel uses the `vm_area_struct` structure, shown in Listing 4.1, for the memory-mapped files. The structure contains the address range of the mapped memory (`vm_start` and `vm_end`), its permission (`vm_page_prot`), and a pointer to the file (`vm_file`). For the sake of the demonstration I omit other fields less relevant to our exploit.

Listing 4.1: Declaration of `vm_area_struct`.

```
struct vm_area_struct {
    unsigned long vm_start;      /* mmap() retval. */
    unsigned long vm_end;
    ...
    pgprot_t      vm_page_prot; /* pg. permissions */
```

```
    ...
    struct file*  vm_file;        /* victim field */
    ...
};
```

Discovery. SCAVY automatically discovers that corrupting `vm_file` can impact the content of the corresponding memory-mapped file, potentially causing privilege escalation. Specifically, SCAVY first creates two executions access the same file (so that they will access `vm_area_struct::vm_file`). Then, it corrupts the `vm_file` with a random value in one of the executions. SCAVY compares the two executions (i.e., with and without the corruption) to discover the two executions obtain different contents of the file.

Next, SCAVY creates the third execution to check whether it can achieve privilege escalation. This time, instead of a random value, SCAVY uses other instances of valid values of `vm_area_struct::vm_file`. Specifically, SCAVY opens a set of both privileged and unprivileged files to obtain the values of `vm_file` instances. SCAVY copies the values to corrupt the `vm_file` and checks whether the third execution can access contents of any files that their `vm_file` values were copied. With careful corruption by copying the existing content of `vm_file` instances, SCAVY analyzes whether the corrupted executions have any changes in accessing privileged resources.

SCAVY checks whether copying the content of `vm_file` from a privileged file to adversary-owned file would allow the adversary to *access the privileged file's content* using the adversary-owned file's permission. For example, if an adversary copies the

`vm_area_struct::vm_file` of the password file (i.e., `/etc/passwd`) to an unprivileged temporary file's `vm_area_struct::vm_file` (e.g., `/tmp/file`), the adversary can read and write the password file. To this end, SCAVY identifies `vm_area_struct::vm_file` as a memory target.

Completing the Exploit. We use `CVE-2022-27666`, which provides an out-of-bounds write capability [61], to corrupt the memory target. The exploit first opens two files: (1) a dummy file with a read/write permission and (2) the password file (`/etc/passwd`) with a read permission. Then, it creates a few thousand memory maps of the files using `mmap` (e.g., 3,000 times in this example). Note that each `mmap` call results in allocating an instance of `vm_area_struct` in kernel. We then leverage the vulnerability to leak *neighboring pages* of the created structures to read the `vm_file` that maps `/etc/passwd`. Next, we use the vulnerability again to copy the content of `vm_file` of the `/etc/passwd` into the dummy file's `vm_file`.

Finally, the exploit calls `msync` with the corrupted `vm_area_struct::vm_start`. This makes the kernel synchronize the mapped page with the content of `/etc/passwd`, using the permissions of the dummy file which is the read and write permissions. To this end, the attacker can add a new root-level account by modifying the `/etc/passwd`.

4.2.2 Corrupting `key::description`

Target Kernel Structure. The `keyctl_instantiate` system call allocates the `key` structure in the kernel, shown in Listing 4.2. It contains fields to store permissions (`perm`), the owner's identifiers (`uid` and `gid`), and a text description of the key

(description). This time, we target description, which is a string pointer where its text is used by the keyctl_search system call which allows a user to search a keyring. The kernel uses the descriptive text to find the correct key when the user issues a search keyctl_search system call.

Listing 4.2: Declaration of key.

```
struct key {
    refcount_t                  usage;
    ...
    kuid_t                      uid;
    kgid_t                      gid;
    key_perm_t                  perm;
    ...
    unsigned long               len_desc;
    char*                       description;
    ...
};
```

Discovery. SCAVY discovers that once the description field is corrupted, an attacker can call keyctl_describe to read a string value from the corrupted address, ultimately allowing the attacker to specify any arbitrary kernel address to read from. Specifically, SCAVY creates two processes which call (1) add_key() with a crafted description including payload to create a key and (2) keyctl_read() to access the created key. In one process, SCAVY injects a corruption that overwrites

`key::description` with a value from another instance of `key`. The other process runs without any memory corruption, and indicates a normal flow of the program. At the end of both executions, SCAVY compares the return buffers of `keyctl_read()` to detect the deviation, discovering a privilege escalation on the kernel memory, which is inaccessible to unprivileged user processes.

Completing the Exploit. We modified an existing exploit for `CVE-2016-0728`, that overwrites `key::key_type::revoke`. The field is a function pointer; thus it allows an adversary to hijack the control flow. However, in practice, this made the exploit unreliable in Android devices, either due to lack of kernel symbols [62] or SMEP/SMAP [63].

Instead of corrupting the function pointer, we focus on `key::description`. As with the original, our exploit first triggers the vulnerability by overflowing `key::usage` to '`0`'. This field is a reference counter, used by the kernel's garbage collector to free unreferenced memory, effectively freeing the key structure prematurely and causing a use-after-free later in the exploit. We use this primitive by allocating a '`msg_msg`' object, which includes a buffer where we can insert arbitrary data (a message). The buffer overlaps with the fields of the `key` object and allows us to overwrite the length (`len_desc`) and the description pointer (`description`), which is the target. If done properly, this allows an adversary to read from arbitrary kernel addresses. With this capability, the adversary can bypass KASLR, and read secret keys from other Android apps' keyrings, potentially leaking session cookies.

4.3 SCAVY in the Kernel Exploitation Development Pipeline

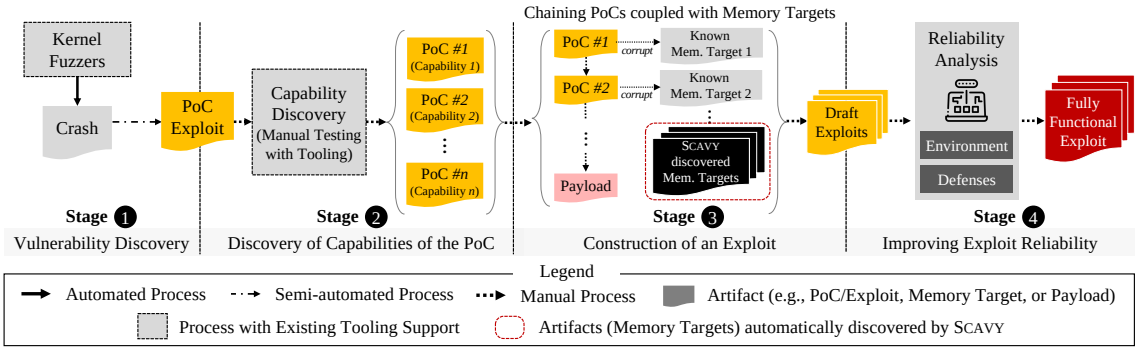

Figure 4.2: SCAVY in the Linux kernel exploit development pipeline

In this chapter, I will contextualize our work with respect to the prior research on the Linux kernel exploitation development. I provide an overview of the line of research on the Linux kernel exploitation development and contextualize our work with respect to the prior research.

Kernel Exploitation Development Process. As shown in Figure 4.2, a functional kernel exploit is created through *four stages*: ❶ Identifying a vulnerability causing a crash, ❷ Discovering the vulnerability's other capabilities that can corrupt various memory targets, ❸ Combining the capabilities to escalate privilege, and ❹ Creating a reliable exploit escalating privilege while bypassing defenses.

— **Stage ❶**. Vulnerability Discovery. —

Fuzz Testing Approaches. To find a vulnerability that can change a system's behavior, various testing techniques, such as fuzzing, have been proposed. Syzkaller [64]

is a popular fuzzer that is specialized for finding Linux kernel vulnerabilities. Recently, various advanced fuzzers have been proposed, including those leveraging hybrid fuzzing [65], symbolic execution [66], and state-based exploration [67–69] to improve the effectiveness of testing for vulnerability discovery.

Exploiting Violation Detectors. Another notable advancement in vulnerability finding is the use of violation detectors, such as sanitizers. Violation detectors [70–74] are runtime techniques that raise exceptions when they detect operations violating desired properties (e.g., out-of-bounds reads/writes [70], use-after-free [71], and race conditions [72, 73]). When they are applied to the target system, they essentially turn the violations into crashes, helping fuzzers identify the violations that can be a strong indicator of potential vulnerabilities. Additionally, they are an attempt at unifying various crashes into a common route cause, effectively narrowing the distance between the crash and the violation that caused it. Recently, leveraging such detectors has become a typical tactic in the Linux kernel fuzzing [74].

Relevance to SCAVY. An exploit requires a vulnerability that can corrupt a specific memory, which can lead to privilege escalation. A vulnerability in this stage typically causes a crash, often because it corrupts a critical memory. However, it is unclear whether it can corrupt a specific memory target with a desired value. SCAVY does not attempt to find bugs; instead, it assumes that a bug and a way to trigger it to a crash have already been found.

In subsection 4.2.1, this step is equivalent to choosing a known vulnerability (i.e., `CVE-2022-27666`,`CVE-2017-7308`).

— **Stage ❷**. Capability Discovery. —

Capability of a Vulnerability. A vulnerability is often released with a *single memory corruption target*, which typically causes a kernel crash when triggered. However, such an initial capability may not be sufficiently powerful and versatile enough to achieve privilege escalations. As a result, investigating a vulnerability's other capabilities has become a critical step to see if the vulnerability can be exploitable. A line of research exists that searches a vulnerability's full capabilities [13–15]. Specifically, [13] introduced capability-guided fuzzing to investigate out-of-bounds write vulnerabilities. [14] leverages fuzzing and symbolic execution to explore various contexts of a use-after-free (UAF) vulnerability and determine if the attacker can control the system to reach an exploitable state. [15] explores multiple crashes of the same bug in an effort to observe more exploitable crashes. AlphaExp [75] discovers fields of kernel structures that can be exploited to achieve arbitrary code execution (ACE) or arbitrary address writing (AAW) capabilities but not privilege escalation. It focuses on a different set of structure fields, missing structures such as `vm_area_struct` that SCAVY found.

Relevance to SCAVY. This stage explores whether the vulnerability can corrupt a more diverse memory range. As SCAVY discovers memory targets which are essentially kernel data structures' fields, capabilities of are vulnerability is critical to see whether it can be used to corrupt the SCAVY's memory targets. To use a SCAVY's memory target, a vulnerability should have a capability that can corrupt the memory target.

In subsection 4.2.1, the vulnerability's exploit *already has the capability* of corrupting the memory target.

— **Stage ❸**. Construction an Exploit. —

Escalating Privilege. Privilege escalation is considered the holy grail in kernel exploitation. As such, it is usually the end goal of capability chaining. Chaining multiple capabilites from one or more bugs is a crucial step in a local kernel exploit, as it allows the attacker to bypass defenses up to gaining full control over the target system. Privilege escalation is typically achieved either (1) by gaining *arbitrary code execution* or (2) by *elevating privilege of the current user/resource* to root.

First, existing approaches [13, 14, 76] automatically test whether a vulnerability can corrupt a known field of data structures that can hijack the control flow (e.g., function/data pointers). Second, existing scripts [77] search for fields of kernel structures related to known critical system configurations (e.g., `uid`, `gid`, or credentials [5]) that can lead to privilege escalation if corrupted. However, they focus on *known fields* or *limited types of fields* (e.g., code or data pointers [13, 14, 76] and *reference counter* [78–80]). While approaches searching for other types of memory targets [78–82] exist, they are either manually done or focusing on certain types. For example, [81] focuses on structures handling variable-length data, which can allow out-of-bound reads if corrupted and [82] looks for structures with data pointers, which allows adversaries to control the referenced data if corrupted.

Some exploits require *chaining multiple vulnerabilities (or PoCs)* to corrupt multiple memory targets (e.g., leaking a value first and corrupting a field with the value), achieving privilege escalation. Finally, a payload is followed to achieve the exploit's ultimate goal (e.g., taking over the system).

Relevance to SCAVY. Typically, only the vulnerabilities capable of corrupting *a few known memory targets* are used to construct an exploit. Vulnerabilities that can corrupt other memory targets but not those known ones were considered usable. SCAVY discovers new memory targets, enabling more vulnerabilities to be usable for an exploit. SCAVY found memory targets allow an exploit to achieve privilege escalation in more diverse and subtle ways.

In subsection 4.2.1, the vulnerability is used to corrupt a new SCAVY found memory target, `vm_area_struct::vm_file`, achieving the privilege escalation of a single file, instead of escalating an entire process's privilege [83]. Note that we chain two capabilities to first *read* the value of `vm_file` and then *write* the leaked value to another `vm_file` field.

— **Stage ❹**. Improving Exploit Reliability. —

Consistently Corrupting Memory Targets. While an exploit is created in the previous stage, it might not reliably achieve privilege escalation, due to the randomness of memory layout and timing during the execution of the exploit. Hence, exploit authors leverage various techniques [76, 84, 85] to achieve an environment for reliable memory corruption. Specifically, [85] shows a method called *heap feng-shui* that extends to all slab caches and thus can be recycled for multiple kernel exploits. [76] automatically finds system calls to allocate objects of interest in a desired heap memory layout. [84] tests commonly known heap layout manipulation and exploit stabilization methods to aid the exploit development.

Bypassing Defenses. A reliable exploit should also evade existing defenses [10,

86, 87]. To this end, researchers have proposed various techniques to bypass the defenses. Specifically, [88] chains kernel-side ROP gadgets to bypass modern control flow integrity-based protections (e.g., CFI [10]). [89–91] present methods to bypass the Kernel Address Space Layout Randomization (KASLR) [86]. [87] and [92] have proposed a method to bypass Supervisor Mode Execution Prevention (SMEP) [93] and SMAP (Supervisor Mode Access Prevention), respectively. [83] presents practical tricks to improve the reliability of an exploit.

Relevance to SCAVY. While SCAVY is not directly relevant to this stage, using some SCAVY found memory targets makes it easier to achieve reliability. For example, an analysis of the published PoC exploit for `CVE-2016-0728` mentions that existing defenses such as SMEP/SMAP [9, 93] can throttle the success rate of the exploit [63], where some SCAVY found memory targets can avoid the detection of those defenses.

In subsection 4.2.1, SCAVY found memory target can achieve privilege escalation without violating the integrity of code pointers, without requiring a defense bypass.

4.4 Related Work

4.4.1 Searching for the exploitable state

Several research papers have proposed methods to facilitate the exploitation of a found bug. FUZE [14] specifically facilitates the exploitation of use-after-free bugs. In the paper, the authors assume the system starts with a known proof-of-concept program that was found through the conventional kernel fuzzer, Syzkaller.

Based on the bug type the authors propose a method that relies on automating the state search within the time between the occurrence of a dangling pointer and its dereference. To efficiently search the system call space the authors stop the execution of PoC right before the dereference and let a fuzzer explore other system calls. This way the system can reach into new execution states, also named as contexts by the authors. The premise is that the new execution contexts will reach new states into the weird machine. After the system has reached a new crash point the authors employ symbolic execution to analyze its exploitability. They consider all the memory of the freed object as symbolic and execute until the instruction pointer becomes symbolic or a memory move operation is moving symbolic data into a symbolic address. The use of both fuzzing and symbolic execution allows the authors to use the best of both worlds. Instead of being doomed by the state explosion in the early stage they leverage fuzzing to explore new crash sites and then analyze the contexts if they can reach exploitable states using symbolic execution. Similarly, in KOOBE [13] the authors rely on a 2-stage search of fuzzing and then symbolic execution. Unlike FUZE, they detach the search from exploit state detection. Basically, this is the first paper that proposes a way to compare out-of-bound capabilities. The authors consider an out-of-bound capability to be larger if it contains less constrains than the previous one or if the OOB write is larger. Then they modify the Syzkaller to prioritize the execution of programs that have shown new OOB capabilities to a certain extent, without sacrificing on the exploration provided by code coverage signal. When new programs are found to have larger capabilities they then analyze the program to make sure that the capability can be used multiple times or if previous

programs' capabilities can be composed into one another, using a greedy algorithm that stitches the capabilities from the largest to smallest. As a last step, similar to FUZE, they employ symbolic execution to evaluate if the overwrite constrains required by the overflow target objects are satisfied by the constrains of the set of capabilities found by the greedy algorithm. The authors create a symbolic memory where the vulnerable object and target object are side by side and symbolically search through the capabilities to find a solution. Once it's found they expect that the combination of capabilities will provide enough memory corruption capabilities that the attacker can perform kernel-context code execution. Similar to prior work I will rely on fuzzing to explore the search space and the symbolic execution to detect if the new state is close to being exploitable.

4.4.2 Object tracing

When systematically analyzing the exploitable states that a corruption to a kernel object can yield, one needs to be able to track all the object types that get allocated as well as the object types. Lin et al. [15] use static analysis to detect the point where the PoC code crashes and using backward-slicing they find all the allocated kernel objects and their types. Wu et al. [14] design a dynamic tracing mechanism to identify the address of the vulnerable object, however they cannot identify the object type. Additionally, a static analysis on the whole kernel, such as in Static Value Flow (SVF) [94], the indirect call resolution thus requires tracking functions, variables, and all other object, rendering the analysis unscalable to the

large-size Linux kernel. All these papers have in common the fact that they start from analyzing a single proof-of-concept program that was deemed to exhibit unsafe behavior. However, in my proposed work I don't have a starting PoC code making this approach more difficult when doing a large-scale systematic analysis of all kernel structures.

4.4.3 Static analysis on Linux kernel

Coccinelle [95] is a tool that identifies pre-defined bug patterns using text pattern matching on symbolic representations of bug cases. It focuses on intra-procedural analysis. Wang et al. built on Coccinelle to create another pattern matching-based tool that detects potential double-fetch vulnerabilities in the Linux kernel [96]. Sparse [97] identifies problematic pointer use between different address spaces, such as kernel or userspace. Later, Sparse was used to develop Smatch [98], a static analysis framework for detecting various kernel bugs. However, Smatch also relies on intra-procedural analysis and can only detect shallow bugs, and not any deep corruption that can reach exploitable states.

Double-Fetch [99] and Check-it-again [100] are focused on detecting time-of-check-to-time-of-use (TOCTTOU) bugs. Dr. Checker [101] analyzes Linux kernel drivers with a modular design that allows for easy plug-ins of new bug detectors. KINT [102] uses taint analysis to detect integer errors in the Linux kernel, while UniSan [103] uses the same analysis to detect uninitialized kernel memory leaks in userspace. Chucky [104] analyzes missing checks in different sources of userspace

programs and Linux kernel using taint analysis, but it can only handle kernel file system code due to a lack of pointer analysis. These works rely on a type-based approach to resolve indirect call targets, which is not accurate and may lead our analysis of bugs into wrong object types, causing false positives.

MECA [105] is an annotation-based static analysis framework that identifies security rule violations in Linux. APISan [106] aims to find API misuse by analyzing existing codebases and performing intra-procedural analysis to identify bugs. To achieve this, APISan uses relaxed symbolic execution techniques. I expect to use symbolic execution only after the kernel has been led into an unsafe state for final analysis to determine if the exploitable state can be reached.

4.4.4 Directed fuzzing

Directed fuzzing is a method of to fuzz testing, where the test inputs are specifically crafted to exercise particular code paths or target specific areas of interest. This contrasts with traditional fuzzing methods, which generally aim to cover as many code paths as possible without specific targeting. Directed fuzzing is particularly useful for exploring deeper into the weird machine of a bug. In this section, we discuss significant contributions to directed fuzzing, with an emphasis on applications to the Linux kernel.

Driller [107], combines fuzzing with concolic execution to systematically generate inputs that explore deeper code paths. This approach allows for targeted exploration of the code, making it particularly effective in finding bugs in critical

code regions that standard fuzzing might miss. Hawkeye [108] introduces a method for directed greybox fuzzing that focuses on improving the precision of directed fuzzing techniques. It uses a lightweight path-tracking mechanism and a novel mutation strategy to guide the fuzzer towards specific target sites within the code. DIFUZE [109] targets device drivers, which are critical components of the Linux kernel often susceptible to vulnerabilities. By leveraging static analysis, DIFUZE identifies and targets specific interfaces between the kernel and its drivers. This targeted approach enables it to efficiently generate test cases that exercise driver code, uncovering vulnerabilities that would be missed by more general fuzzing techniques.

KOOBE [13], uses directed fuzzing starting from a PoC code that triggered an out-of-bound write and tries to explore other memory layouts to use this OOB. FUZE [14], employs a directed fuzzing method to explore the weird machine states stemming from an UAF bug. Specifically, they use this method to explore the search space between the dangling pointer and the dereference of the pointer. Grebe [15], uses an object-driven kernel fuzzing. Given a kernel bug, the object-driven kernel fuzzing method helps security analysts better understand and infer its exploitability. SyzScope [110], combines fuzzing, static analysis, and symbolic execution together to reveal high-risk security impacts of seemingly low-risk bugs. The directed fuzzing method allows the system to use the same bug to explore different states and find new crash sites in the hopes to obtain more severe crashes.

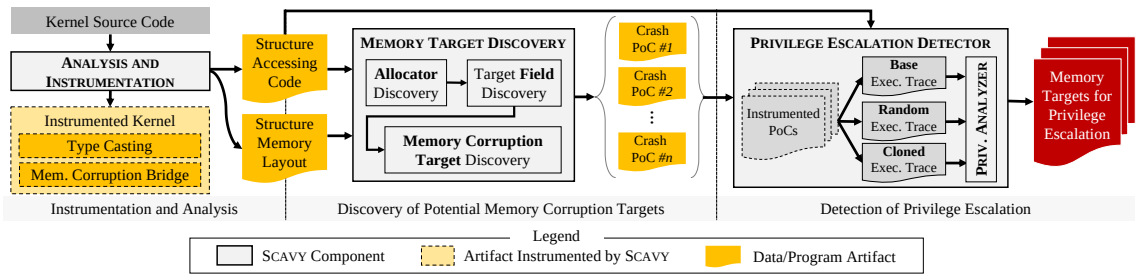

Figure 4.3: Overall SCAVY Design

4.5 Design

Figure 4.3 shows an overall procedure of SCAVY, consisting of three phases: (1) Instrumentation and Analysis (subsection 4.5.1), (2) Discovery of Potential Memory Corruption Targets (subsection 4.5.2), and (3) Detection of Memory Corruption Targets for Privilege Escalation (subsection 4.5.3). Each phase addresses challenges imposed by the new definitions of memory targets and exploitable states in SCAVY. In the following sub-sections, I describe each phase in detail.

In this section, I elaborate on the design of SCAVY. Specifically, I define the search space of the memory targets, including the definition of the exploitable state. I also describe our design choices and the reasoning behind the decisions. Figure 4.3 shows the overall procedure of SCAVY. I define the search space and the design decisions made to satisfy the requirements for effectively searching and detecting exploitable states. I highlight tradeoffs for satisfying our requirements and the differences they present with regard to prior work. I present the design of SCAVY starting from the second step of the pipeline, the *search* step, illustrated in

Figure 4.3.

4.5.1 Instrumentation and Analysis

Type Casting Instrumentation. To identify memory corruption targets, SCAVY needs to identify types of allocated memory (e.g., types of structures) in the kernel and corrupt them. While previous techniques such as GREBE [15] rely on expensive taint analysis, they cannot be used for the sheer number of potential memory corruption targets SCAVY deals with. To this end, SCAVY instruments type casting operations.

Note that the Linux kernel coding-style guideline document [111] indicates that a memory allocation should be type-casted. In LLVM, it uses `CastInst` [112] for the type casting operation. Hence, we instrument `CastInst` by inserting a `call` to a dummy function that takes two parameters: (1) the memory address of the variable that is being type-casted, and (2) its new data type, passed as a string at compile-time. Note that we only instrument `CastInst` when its source and destination types are different (e.g., '`void*`' is assigned to a structure). At runtime, we use Kprobe [113] to log the dummy function's parameters to user space along with the return value of memory allocators (e.g., `kmalloc()`). Later, the fuzzer associates allocated memory addresses with their data type.

Analysis for Kernel Data Structures. During the instrumentation and analysis process, SCAVY aims to extract memory layouts of kernel structures of interest, so that it can guide the subsequent analyses of SCAVY. While we can extract them from

the source code via an LLVM pass, they may not be accurate if a compiler optimizes the memory layouts of the structures. Specifically, structures that are not packed (i.e., structures without the '`((packed))`' attribute) may have unused memory space between fields, making the offsets of structures' fields in a binary different from the structure's definition in the source code. For example, if a structure has a 6 bytes of character array followed by an integer (i.e., `struct { char s[6]; int n; }`), a compiler may insert two unused bytes between the two fields (i.e., after `s[6]` and before `n`, resulting in `struct { char s[6]; char unused[2]; int n; }`). Hence, we use `pahole` [114] to find the sizes and offsets of structures at the binary level. We use construct a lookup dictionary mapping between a structure's name and the structure's fields' offsets, sizes, and types. Note that structures inside a structure are resolved recursively.

Memory Corruption Bridge. Most of SCAVY's components run in user mode, which do not have direct access to the kernel memory. Hence, we implement a kernel module that allows SCAVY's user mode components to corrupt the kernel memory. Specifically, the kernel module exposes an interface via `ioctl()`, providing read and write capabilities of the kernel memory to the user mode programs. During the instrumentation process, we include the bridge module into the kernel.

4.5.2 Discovery of Potential Memory Targets

Three Stage Operations of Exploits. We observe that most exploits in practice exhibit a pattern of three distinctive operations: (1) *allocating* a resource associated

with a memory target, (2) *corrupting* the memory target to obtain sufficient permissions for privilege escalations, and (3) *conducting privileged actions* or actions escalating privilege.

```
1  for (int i = 0; i < 100000; i++)
2      open("/etc/passwd", O_RDONLY);
3  int wf = open("/tmp/writablefile.txt", O_RDWR);
4
5  CORRUPT_FILE_MAPPING(/* SPRAYED OBJ. MAPPING */);
6
7  write(wf, "root2:x:0:0:root2:/:/bin/bash\n", 32);
8  close(wf);
```

❶ ❷ ❸

(a) Exploit that Corrupts File Mapping

```
9  int pid = fork();
10 if (pid == 0) {
11     CORRUPT_euid(0);
12     exit(0);
13 } else {
14     waitpid(pid);
15     setuid(0);
16     execve("/bin/sh");
17 }
```

❶ ❷ ❸

(b) Exploit that Corrupts **`euid`**

Figure 4.4: Exploits Consisting of Three Steps

Figure 4.4-(a) shows the exploit described in section 4.2. It first creates two files. The memory targets are associated with the files (❶). Then, it corrupts the memory target, which is a kernel structure storing information of the file via the vulnerability (❷). This allows an adversary to obtain the permission of the root-owned file '`/etc/passwd`.' It then adds a new user by writing the '`/etc/passwd`' file, escalating privilege (❸). Figure 4.4-(b) follows a similar procedure. It creates a process that allocates the memory target `task_struct` (❶), and then corrupts

the '`task_struct::euid`' associated with the created process, changing the process's effective user to the root user (❷). Finally, it creates a privileged shell via `execve()`.

Search Space for Each Stage. As each stage conducts a different operation, candidate operations for each stage may exist in a different space, requiring a distinctive focus. Specifically, to search operations for the first stage, one may look for system calls that *allocate* memory buffers, regardless of whether they will access the buffers or not. For the second stage, one should focus on the contents of the memory buffers and the impact of the corruption on the system. For the last stage, we should focus on the system calls dependent on the corrupted memory (i.e., system calls using the corrupted data).

Due to, in part, the different focus on each stage, in practice, different tools are used. For example, Syzkaller [64] is effective for searching the first stage because it aims to achieve higher code coverage. Intuitively, covering more code would exercise more program paths that may allocate data structures. For the second stage, KOOBE [13] and FUZE [14] are effective as they implement heuristics for detecting memory corruptions, such as overflowing into other co-allocated objects. Unfortunately, there are no popular tools for the third stage as it is manually done in practice.

SCAVY's Approach. Unlike existing tools that are only effective for each stage, SCAVY proposes to apply different criteria and metrics for each stage dynamically. Specifically, for the first stage, we use code coverage of system calls as a metric. Intuitively, by covering most of the code, it might also cover all the kernel structure allocation code of interest. For the second stage, we use coverage of instructions that

load the corrupted memory to identify whether the corruption has an impact on the system or not. In the third stage, we focus on code coverage of the code dependent on the corrupted memory, to explore diverse consequences of the corruption.

4.5.2.1 Allocator Discovery

Objectives. This stage aims to identify a list of system calls that allocate kernel structures, which are the memory corruption targets. We aim to find system calls allocating as many unique kernel structures as possible because all the subsequent analyses are limited to the result of this stage. SCAVY runs every system call, tracks accesses to all allocated memory buffers, and detects types of allocated kernel structures.

Tracking Allocated Structures. We track memory allocations and memory accesses to identify allocated kernel structures as follows. We use Kprobe to set breakpoints on `kmalloc()`[1] and `kmem_cache_alloc()` to capture their return values (i.e., base addresses of allocated kernel structures).[2] We also set a breakpoint on the dummy function in subsection 4.5.1 to capture its arguments. Note that analyzing type casting (and not using expensive taint/symbolic analysis [15]) is a key design choice that makes SCAVY scalable, allowing us to conduct the analysis on millions of generated sequences.

Coverage Guided Searching. SCAVY tries to cover more code allocating kernel structures, aiming to discover allocators for diverse memory targets. We use code

[1] While it is `__kmalloc()`, we simply call it `kmalloc()` hereafter.

[2] We track `kmalloc()` and `kmem_cache_alloc()` in the process of interest.

coverage, *without changing the fuzzer and the coverage*, as, intuitively, covering more invocations of `kmalloc()` and `kmem_cache_alloc()` would likely find allocators for various memory targets.

4.5.2.2 Memory Target (Structure Field) Discovery

Objectives. Corrupting certain parts of the kernel structures change the system's behavior, while other parts may not have an observable impact on the system. Hence, SCAVY aims to find which fields of the structures can be memory targets causing an observable impact. Specifically, among all the allocated structures in the first stage, we search, one field at a time, which fields of structures can impact the system when corrupted (e.g., a crash). Similar to subsubsection 4.5.2.1, we search conservatively, as the subsequent analyses depend on it.

Conservative Target Searching. We use a *conservative* definition that *if a corrupted field of a kernel structure is read at least once, it may impact the system*, meaning that it is a potential memory corruption target. Specifically, we corrupt each 4-8 byte field of the kernel structures with random bytes and use the `hardware watchpoint` [115] to monitor instructions that load the corrupted field. If we identify any such *load* instructions, the field is considered a potential memory target, and SCAVY moves on to the search for the next field. The `watchpoint` is lightweight and reports sufficient information on the memory access (i.e., whether the field was accessed).

4.5.2.3 Memory Target Discovery

Objectives. With the potential memory target for structures, SCAVY uses a fuzzer to execute various system calls that may access corrupted memory and cause a crash. In this stage, we *prioritize* code that *accesses the corrupted memory* (i.e., the corrupted field of structures). However, during fuzzing, there can be crashes (i.e., kernel panic) on the first access of the corrupted memory, which we call *premature crashes*. Those crashes may prevent the fuzzer from exploring the full impact of the corruption, missing later corrupted memory accesses.

Focused Fuzzing on Relevant Code. To facilitate the search in this stage, we make two adjustments on our fuzzing process. The adjustments are based on our consideration of what is important code for the fuzzer to cover. The important code is the one that (1) operates on the structure type from stage 2 or (2) the code that *loads* from the corrupted field in that struct.

Therefore, as the first adjustment, we make it focus on the code that is accessing the kernel structures of interest. Specifically, the fuzzer is given a list of the kernel functions that operate on each `struct`, and generates a coverage filter at runtime if that `struct` is being corrupted. At runtime, the fuzzer narrows down the scope of the coverage to the code that accesses the kernel structures of interest.

Second, we prioritize the instructions loading the corrupted memory by detecting executions running the instructions and seed them. Specifically, recall that SCAVY collects information about the code related to the structures of interest in the instrumentation and analysis phase, as shown in Figure 4.3 (Structure Access

Code). SCAVY uses hardware breakpoints[3] to detect those instructions and make them seed.

Avoiding Premature Crashes. Normally, if the kernel access the corrupted field and finds an unexpected value it will crash, causing the fuzzer to miss other syscalls that might lead the kernel to exploitable states. We propose an approach that executes a generated program call twice to avoid premature crashes. We solve this problem by modifying the execution step. Instead of just executing the whole program from stage 1 to 3, we first change stage 2 to only set the HW breakpoint at the field of interest without corrupting it. We execute stage 3 as normal and collect all the coverage and accesses to the uncorrupted field of interest. Then, we re-execute the same program where the stage 2 corrupts the field of interest. The fuzzer will get the code coverage as above to guide its search, while also detecting the crash and reporting it.

4.5.3 Detection of Privilege Escalation

The outcome of subsection 4.5.2 is a list of programs causing crashes, some of which may cause privilege escalations. In this phase, we aim to detect which of them from the previous phase are causing privilege escalation. Specifically, SCAVY uses a differential-analysis-based approach to detect privilege escalation as follows. First, we run a given program thrice: one run without corruption, one with corruption using a random value, and another with corruption copying the value of the memory

[3]Since hardware breakpoints require addresses of the instructions, we use `addr2line` [116] to obtain the corresponding instruction addresses.

target from other valid structures created by a privileged process. Second, all three runs will execute privileged operations (e.g., operations requiring root permission) such as `setuid(0)` and `read`/`write` on privileged resources. If the first and the third runs have differences, we consider it may have escalated or de-escalated privilege. In other words, we leverage our definition of the exploitable state to mutate the DoS exploits to reach the exploitable states. In prior work, this search step is done using symbolic execution [13,14,117]. However, this method can be computationally expensive, especially when dealing with a large corpus of crashes such as the one produced in the *Search* step. Our fuzzer produced *3863* proof-of-concept (PoC) 3-stage programs. Additionally, many PoCs lacked read or write syscalls in stage 3, so the exploitable state wouldn't be reached. We address both challenges below.

4.5.3.1 Inserting Privilege Dependent Operations

Read and Write System Calls. For each PoC, we insert read and write system calls for all the resources (e.g., files and sockets) created during the PoC, as their behaviors will be different if privilege escalation happened. For example, as shown in Figure 4.4-(a), we add read and write system calls for all the files (i.e., '`/etc/passwd`' and '`/tmp/writablefile.txt`'). This is done to ensure that the corruption causes an operation on the file descriptor to perform privileged actions. This step is not necessary since the fuzzer already access the corruption, but it helps when manually verifying other potential effects of the corruption. In some programs, this step helps the detector flag other exploitable states that are not in the original PoC.

Privileged Operations. To make sure the detector will flag the exploitable state we enrich each PoC by inserting read and write syscalls for all the resources created in stage 1 (ie. files, sockets, etc). For example, in the code at Figure 4.4 we would be adding read/write syscalls do all resources in blue lines (ie. `res[i]`, `vic`). In most of the programs there was only 1 resource opened in stage 1 making it easier to add the new syscalls.

As shown in Figure 4.4-(b), privileged operations behave differently based on the current privilege. We add privileged operations (i.e., root) such as `seteuid(0)` and `setegid(0)`. `sync()`, `msync()`, and `fsync()` are added for all the resources created before the memory corruption. We provide a list of all the added syscalls in Table 4.1 below.

System calls
open("/etc/passwd",O_APPEND)
open("/etc/shadow",O_APPEND)
open("/etc/shadow",O_RDONLY)
setuid(0)
setgid(0)
open("/proc/self/mem",O_RDONLY)
open("/proc/1/mem",O_RDONLY)
opendir("/root")
socket(AF_INET, SOCK_RAW, IPPROTO_RAW)
mount("dev/sda1", "/mnt", "ext4", MS_RDONLY, NULL);
chmod("/etc/shadow", S_IRUSR — S_IWUSR)
kill(1, SIGKILL)
mknod("/dev/mydevice", S_IFCHR — 0600, makedev(10, 100))
unlink("/etc/shadow")
symlink("/etc/passwd", "/tmp/passwd")

Table 4.1: List of system calls that conduct operations dependent on privilege.

4.5.3.2 Detecting Exploitable States

Three Executions for Privilege Escalation Testing. We reason that an exploitable state is reached due to the corruption if the same program shows a deviation in the execution of stage 3 based on whether the corruption is carried out in

stage 2 or not. We use this reasoning and execute the now-enriched PoC 3 times to effectively detect exploitable states while avoiding the computational overhead of symbolic execution. First, we run the exploit on an unprivileged system without memory corruption. We expect all the privileged operations to fail and unprivileged operations to succeed. Second, we run the exploit on the same system with memory corruption using a random value. If privileged/unprivileged operations behave differently from the first execution, we consider it to have reached an exploitable state. Third, we run the exploit on the unprivileged system with memory corruption using the memory contents of another instance of the same type kernel structure. Specifically, we borrow a value of a structure's field from a root process with the same structure. Again, any deviations from the first run suggest privilege (de-)escalation. Note that the random and cloned values are complementary. Cloning works well for complex kernel data structures with certain formats as they are difficult to generate randomly. However, cloned values limited to what the system created. For example, cloning `uid` will never result in an invalid/unknown `uid`, which a randomly generated value can.

Detecting Exploitable States. For each run, we collect 4 indicators: (1) the return value of system calls, (2) the system call's return buffer's (or read buffer's) data, (3) the addresses of instructions loaded the corrupted memory, and (4) the code coverage. We consider two executions to be the same if the following 4 conditions are satisfied: (1) return values of the system calls are identical, (2) return buffers' contents of the system calls are identical, (3) executed `load` instructions on the memory target are identical, and (4) Jaccard similarity [118] of the code coverage

is greater than 0.9 (accounting for noise in Kcov). Note that we have three runs to compare (2 pairs to compare). We compare the run without corruption with the two other runs with different corruption methods. If the run with corruption is different with any of the comparisons are not the same, we consider it to reach an exploitable state.

For example, in Figure 4.4-(a), SCAVY first runs it without a memory corruption (line 5). The `write()` at line 7 will fail due to the missing privilege. In the second run, it corrupts the memory target (i.e., file mapping pointer) with a random value. The execution will crash or fail due to the invalid file mapping pointer. In the third run, it corrupts the memory target by cloning a valid file mapping pointer. Now, `write()` at line 7 will succeed and SCAVY detects differences in code coverage and `write()`'s returns. Similarly, in Figure 4.4-(b), executions have different coverages and `setuid()` return values.

4.6 Implementation

SCAVY is written in Go (2068 LoC), C (331 LoC), C++ (827 LoC), and Python (2199 LoC). We customized the Syzkaller in Go. In C, we wrote the in-tree kernel driver used by the fuzzer to corrupt and set up hardware tracing. We also developed 3 LLVM passes, two of which produce the input to the fuzzer and the third to instrument the kernel typecast. Our privilege escalation detector module is written in Python. Go is used to customize the Syzkaller. In C we wrote the in-tree kernel driver used by the fuzzer to corrupt and set up hardware tracing.

C++ is used to write the 3 compiler passes, two of which produce the input to the fuzzer and the third to instrument the kernel typecasts, as shown in the green box in Figure 4.3. Python is used for the analysis step to parse the program from crash logs, enrich it and run it 3 times, parse its 4 different output types and detect the deviation.

4.6.1 Compiler instrumentation

The input to SCAVY is the modified kernel source. For this paper we chose to use the LTS Linux kernel version *5.15.80*. First, we add an in-tree driver in the `drivers` folder. We also modify the `Kconfig` and `Makefile` in that folder. Then we simply enable `CONFIG_CORRUPTER_MODULE` in the *.config* file before compilation. When making the kernel we change the `CC` to point to a shell script that calls a python script that makes the decision whether the file should be instrumented or not based on its relative path. The instrumented file is ran through 2 of our compilation passes by `clang`'s Xclang plugin with the following command-`fcommon -Xclang -load -Xclang` $<$ *pass.so* $>$ `-flegacy-pass-manager` $<$ *input.c* $>$. The uninstrumented file is compiled with the default clang.

The first pass produces a text file that contains the structure types that each function in the file works with. The second pass instruments every `CastInst` instruction whose source is an `int8*` and destination is a struct pointer. The instrumentation step involves adding a function call to an external function defined in our in-tree custom driver. The function takes as arguments the name of the caller func-

tion (the one we are instrumenting) the pointer value that is being typecasted, and its new type. To protect the function for any optimization step we make it change a global variable to the XOR value of the passed pointer and its caller function name's pointer.

4.6.2 Fuzzing for privilege escalation

Our fuzzer is a modification of Syzkaller. We modified its 3 core components: syz-manager, syz-fuzzer and syz-executor. The manager is modified to load the 2 dictionaries illustrated in Figure 4.3. Once loaded they are sent to syz-fuzzer. The manager also has a parameter for 2 modes of execution. In the total tracking mode it enables all syscalls and logs out all the objects created. The second mode is the normal privesc exploration one.

Modifying syz-fuzzer The `syz-fuzzer` is the core of our modifications. It is the program that generates and mutates the seeds that will be run by syz-executor. We modify its execution step to execute the same program 3 times instead of once as mentioned in subsection 4.5.3. For each execution we collect data from KCOV, Kprobe and *dmesg*. Normally syz-executor collects the code coverage from KCOV, our modification allows `syz-fuzzer` to collect the allocations frees and typecasts from Kprobe and memory hits of corrupted memory from dmesg. To collect data from Kprobe our `syz-fuzzer` first instructs Kprobe to set up its probe points for `kmalloc`, `kmem_cache_alloc`, `kfree` and `instrument_typecast_instruction`, the custom function that does nothing. The fuzzer instructs Kprobe to print all param-

eters of the custom function and kfree and the return value of the two allocation functions. After setting up the *Kprobe* point our fuzzer will enable Kprobe to run right after spawning the executor. Then it will run a thread in parallel to parse the output logs of Kprobe and filter out any logs that are not generated by syz-executor. To parse *dmesg* the `syz-fuzzer` will spawn another thread in parallel.

Another modification to the `syz-fuzzer` is in the way it communicates with executor. If the seed has a corruption after a syscall (ie. at syscall 6) the executor is instructed to pause execution and tell the fuzzer to perform the corruption. This step is useful for execution 2 and 3 where the fuzzer collects memory hits from dmesg and filtered code coverage. For the filtering we utilize the built-in filtering functionality of Syzkaller. The two differences are that we change the filter on the fly for each structure type that is being corrupted and only apply it to coverage collected after the corruption.

Last addition is the comparison of execution 2 and 3. If the 3^{rd} execution, in which the fuzzer overwrites the target field in the kernel object with random data, does not crash its filtered post-corruption code coverage is compared to that of execution 2. If a deviation is detected the fuzzer will `printf` a line starting with *WARNING*, which triggers the default log parsing in `syz-manager` which assumes the kernel crashed and adds the log to the crash corpus.

Modifying syz-executor We first disable the fork server which makes it certain that after executing the requested sequence of syscalls the executor terminates freeing all the allocated kernel objects. Then we modify the way it executes seeds that have a corruption. As mentioned earlier, the executor replies back to the fuzzer

that it has reached the target syscall and waits for another message from the fuzzer to continue with the rest of the sequence until it's done and the coverage it collected from KCOV is sent back as a reply to the fuzzer.

4.6.3 Detecting privilege escalation

After having a directory full of crash folders we need to vet out the ones that are not reaching privilege escalation. In this step we borrowed part of code from FUZE which streamlines the interactions with QEMU. We modified the initial codebase to run in Python 3.6+. Before enriching the PoC there is parser component (*216* LoC Python) we did not draw in Figure 4.3. This parser goes through the directory of crash folders and parses the crash description and every program that was executed and logged. In the case of our synthetic crash we had *83* different logs in the same crash hash, the script parses each log file and makes it easy to access each executed program for the next steps. For each log file there are multiple programs executed before or even after the one that caused the crash since some crashes are not fatal. To accurately find the one that caused the parser uses a combination of regular expressions.

Enriching the PoC After parsing the Syzkaller representation of the syscall sequence, we get the number of resources created before the corruption. As we observed in subsubsection 4.5.3.1 there most of the programs have only 1 resource before the corruption. We insert `read`, `write` and `fsync` for each of the resources. For all the syscalls we print out their return values and the buffer value in case of

a read syscall. These are read by the deviation detection module, the last piece in Figure 4.3.

Adding corruption triggering code Once we have the sequence of system calls we use the `syz-prog2c` utility to generate the C code. However, we also need to add into the generated code the part where the memory corruption is injected. We wrote a library and included it as a `.h` file (*883* LoC C++) in the generated C code. The library spawns 2 threads along with the original running PoC code, one that sets up the same Kprobe reading and the custom driver interacting code we wrote into the `syz-fuzzer` and another thread that runs as root user and synthetically opens the privileged resources. The first thread tracks the objects created by the PoC, the synthetic root process and the native root processes with $uid == 0$, as we also mentioned in subsubsection 4.5.3.1. The library code uses pipes to communicates with the original C code to make sure the corruption is inserted at the same spot it was observed in Syzkaller's logs. It is set up to introduce the least amount of extra code in the generated PoC's code and be easily modified when one wants to execute PoC in different modes. There are 3 execution modes: (1) execute the PoC without corrupting the memory but only track the accesses to the corruptible field, (2) corrupt the field with random data, (3) corrupt the field with data from the same object type allocated by a root-owned process. The 3 modes are ordered to execute and controlled by the deviation detection module, which collects the same execution data for all 3 modes.

Detecting privileged operations and de-escalation By default Syzkaller runs as root, so the PoC is expected to run with the same privileges. If it runs

in user mode most of the syscalls will return -1 and nothing will happen. This is why we have 3 executions for each PoC. Since a random value overwrite will likely change the root values, ie. the uid from 0 to to non-zero, we are able to detect de-escalation. This module is written in Python (*600* LoC) and uses part of FUZE code to spawn and control the QEMU VMs and parse the syscall return values, code coverages and buffer data from the execution of the PoC.

4.7 Evaluation

We evaluate SCAVY from three aspects. First, we measure the coverage of the typecast instructions with respect to all the Linux kernel structures to evaluate the effectiveness of our typecast instrumentation[4]. Second, we evaluate the effectiveness of the fuzzer in exploring crashes that can lead to exploitable states. Third, we analyze the detection results of the differential analysis module and evaluate their reachability on real world exploits. We also compare our exploitable state definition with FUZE's to show that our broader definition helps discover new memory targets. All experiments were done on a system running Ubuntu 20.04 LTS with an 3.70 GHz Intel Xeon E3-1245 v6 and 32 GB of RAM.

[4]If a typecast instruction is not instrumented, SCAVY may not analyze the specific use of kernel structure, possibly missing a memory target.

4.7.1 Typecast Coverage

SCAVY statically instrumented 2,313 source code files (85% of the 2,700). 23% (2,130 out of 9,169) of the kernel data structures are instrumented[5] to track their types after their allocation at runtime. While analyzing the low coverage result, we find many structures that are covered are irrelevant to us. First, there are 2,587 structures for debugging purposes such as `kprobe_insn_cache` and `trace_event_raw_kfree` are used by Kprobe and Ftrace and are mostly unavailable on production kernels. Second, there are 1,989 stack-allocated structures that are destroyed on function return (e.g., `msg_receiver` and `msg_sender` in `do_msgrcv` and `do_msgsnd`). Those short-lived structures typically do not hold critical system states and, hence, are not our target. Third, 1,931 structures are members of our instrumented structures, which we can consider instrumented. For example `signal_struct` is contained within `sigpending` – there is no code that directly allocates it. Fourth, many of the remaining structures are processor-related global variables (e.g., `intel_watermark_params`) and architecture-specific structures (i.e., defined in `arch`) that we do not instrument or helper structures for accessing their other structures (e.g., `xfrm_skb_cb` for accessing `sk_buff->cb`).

To this end, we prune out 4,576 from 9,169 and additionally consider 1,931 structures to be covered, resulting in a new coverage of 88.4% (4,061/4,593). Our coverage can be interpreted as a lower bound of what our fuzzer can corrupt.

[5]We noticed that instrumenting certain boot files caused the kernel to crash. Hence, we restrict the instrumentation to directories other than '`arch`', '`boot`', '`kernel/panic`', and '`signal`'.

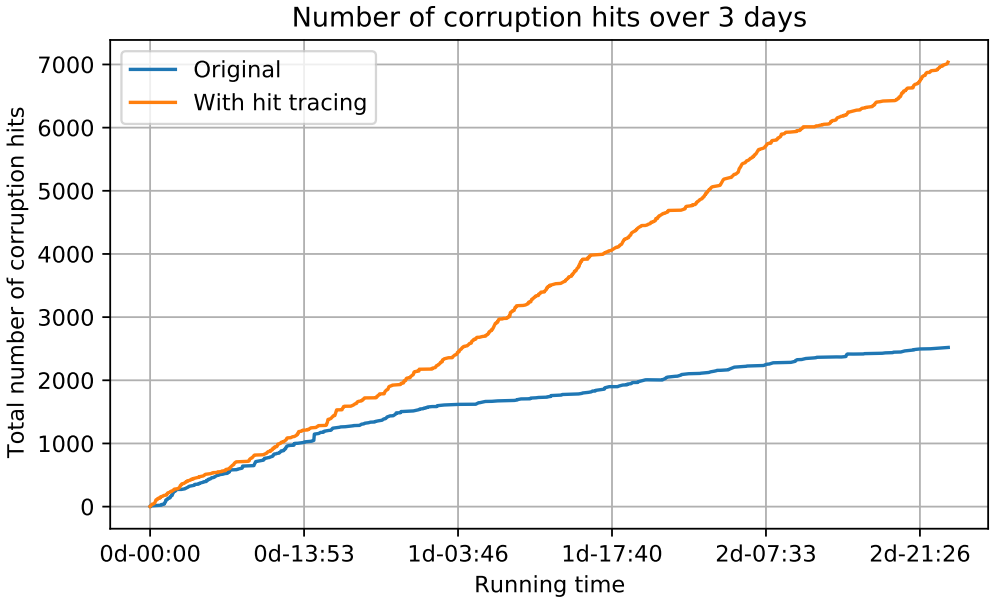

Figure 4.5: # of Observed Loading of the Corrupted Fields

4.7.2 Fuzzer Effectiveness

We evaluate the effectiveness of our fuzzer, compared to the stock version (commit: 61f86278) of Syzkaller. Specifically, we measure the coverage of the code that is relevant to structures (e.g., allocating and accessing the structures of interest).

For a fair comparison, we modify the stock version of Syzkaller by adding hardware tracking of memory reads, our object allocation implementation, and corrupted memory access tracking. We run both fuzzers for 3 days on the identical QEMU VM (with 1 core and 1 GB RAM).

Code Coverage. Figure 4.5 shows the total number of read operations on any corrupted memory during the 3 days of fuzzing. This number indicates the number

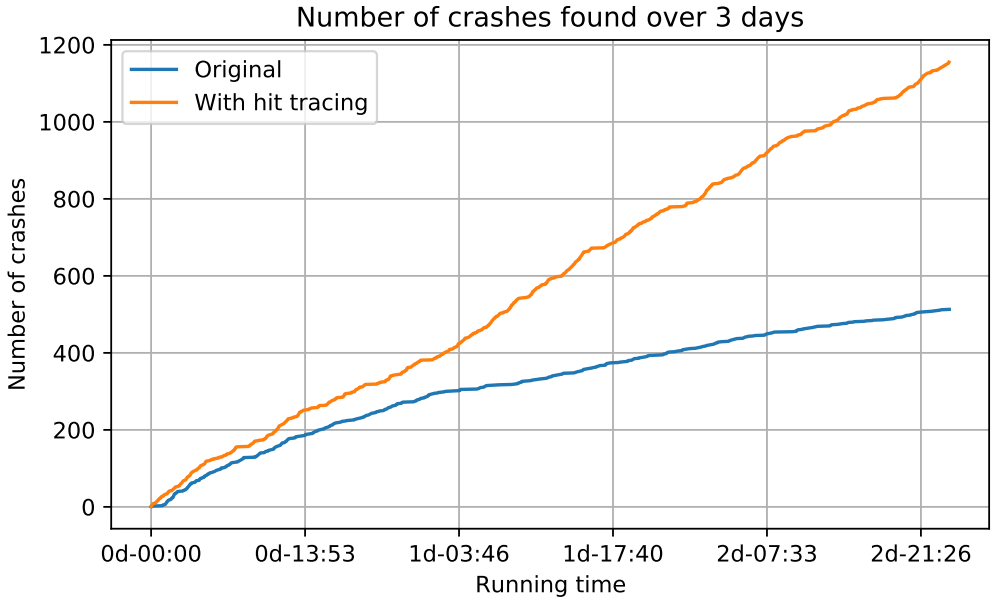

Figure 4.6: # of Crashes Observed

of *instructions loading the corrupted memory* executed during fuzzing. The graph shows that for the first 14 hours, the 2 fuzzers perform similarly, but the stock fuzzer seems to hit the corruption at a slower rate. When manually checking the corpus we find this happens because, in part, the stock fuzzer prioritizes exploring code paths, where some of them may not access the fields of structures of our interest. For example, the stock fuzzer may prefer to open a file and then immediately open a socket, instead of writing to the opened file, since opening a socket can lead to higher code coverage. In contrast, SCAVY prioritizes covering the code that is relevant to the data structures of interest (e.g., accessing the file in the example).

# of Crashed Observed. Figure 4.6 presents the number of crashes observed from the two fuzzers. While both fuzzers inject memory corruptions into the least represented structures, the stock Syzkaller does not prioritize hitting the memory

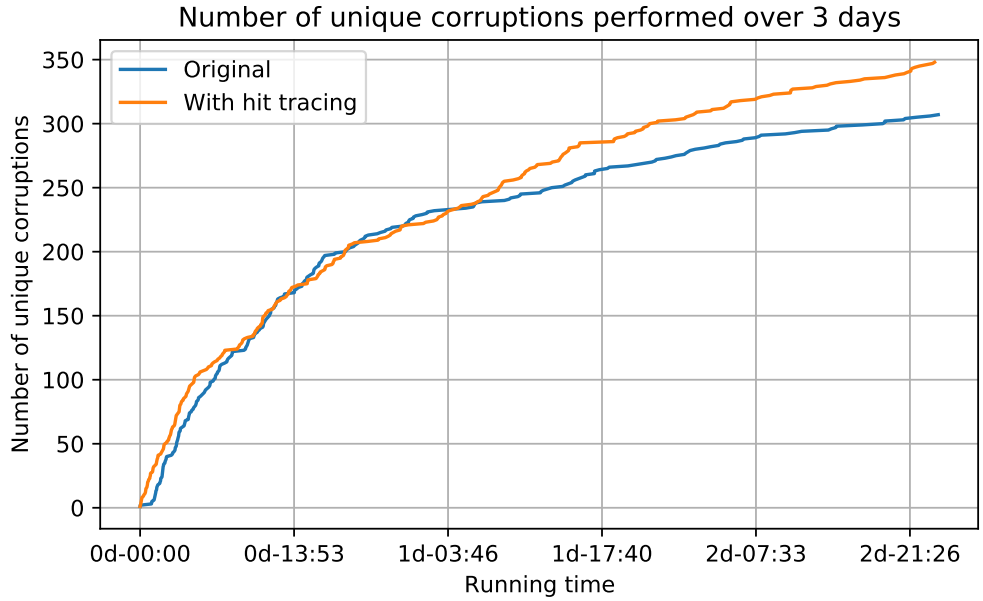

Figure 4.7: Comparison of the Number of Unique Fields Corrupted over Time

it corrupts. This can be inferred by the lower number of crashes, suggesting the de-prioritization of corrupted memory accessing code while prioritizing the overall coverage. In contrast, our SCAVY fuzzer was able to generate more crashes as it prioritizes the instructions accessing corrupted memory. This is further evidence of the effectiveness of our approach, which prioritizes the exploration of the search space for exploitable states; focusing on the code coverage from source code that operates on the target structure type and adding synthetic code coverage based on code that loads corrupted memory.

# of Corrupted Fields. We measure the number of corrupted fields during the experiments to show that SCAVY covers diverse kernel data structures than state-of-the-art techniques. Specifically, we plot the number of unique fields of structures that are corrupted. Figure 4.7 shows that both fuzzers exhibit a similar rate of

exploration in discovering new fields to corrupt.

While SCAVY focuses on a set of structures of interest, this graph suggests that our fuzzer does not become stuck corrupting limited types of structures. As described in subsubsection 4.5.2.1, while SCAVY leverages redefined coverage that may focus on specific structures, our fuzzer still operates with some guidance from unfiltered code coverage, and as a result, it works well for both exploration and exploitation. Note that we repeat the 24-hours experiment 10 times, following the best evaluation practices for fuzzers [119]. We present the extended results in subsubsection 4.7.2.1. After proving distinction between the two distributions using the U-test [41], we conclude that the fuzzer without corruption-hit-guidance tends to cover more code, because, in part, they do not deal with crashes caused by the corruptions. However, the crashes caused by corruptions are potential privilege escalations detected by SCAVY, meaning that the original fuzzers' high code coverage is irrelevant to finding memory targets. Enabling the corruption-hit-guidance leads to a lower code coverage, as it causes crashes more often. but it causes more crashes over time, causing the fuzzer to focus on the code that triggers uses corrupted memory. In subsubsection 4.7.2.1 we show the effects of the modifications of the fuzzer on the code coverage.

4.7.2.1 Comparing with the unmodified Syzkaller

We compare SCAVY's fuzzer against an unmodified Syzkaller baseline using the same methodology described in subsection 4.7.2. We disabled the reproducer and

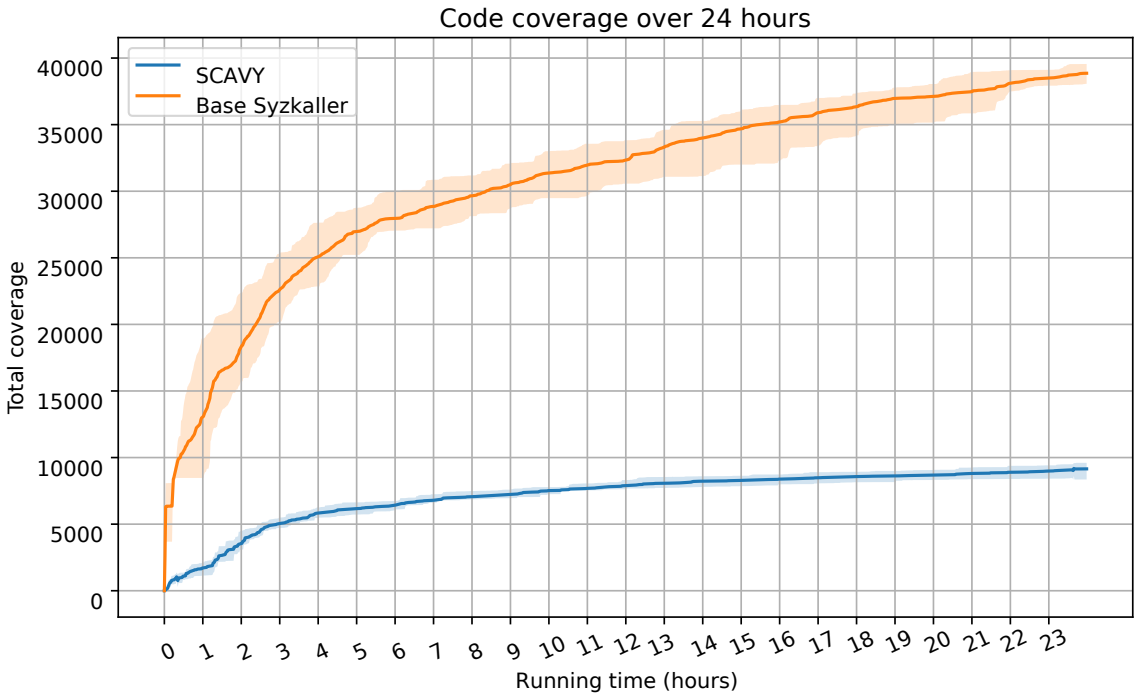

Figure 4.8: Code coverage of SCAVY VS the unmodified fuzzer.

ran it on 1 VM with the debug flag. Figure 4.8 shows that SCAVY's fuzzer achieved a lower code coverage than the original Syzkaller. This does not mean our fuzzer is ineffective, but highlights that SCAVY's fuzzer is more focused on the memory target accessing code rather than exploring irrelevant kernel code.

When comparing the number of crashes found by each fuzzer, we see that SCAVY's fuzzer found more crashes than the unmodified Syzkaller, as shown in Figure 4.9. This is because SCAVY's fuzzer is more focused on the memory target accessing code, which is more likely to cause crashes. Additionally, SCAVY will try to explore deeper into the code that accesses the corrupted memory, which is more likely to reach code that can cause a crash. These crashes are potential privilege escalations detected by SCAVY, meaning that the original fuzzer's high

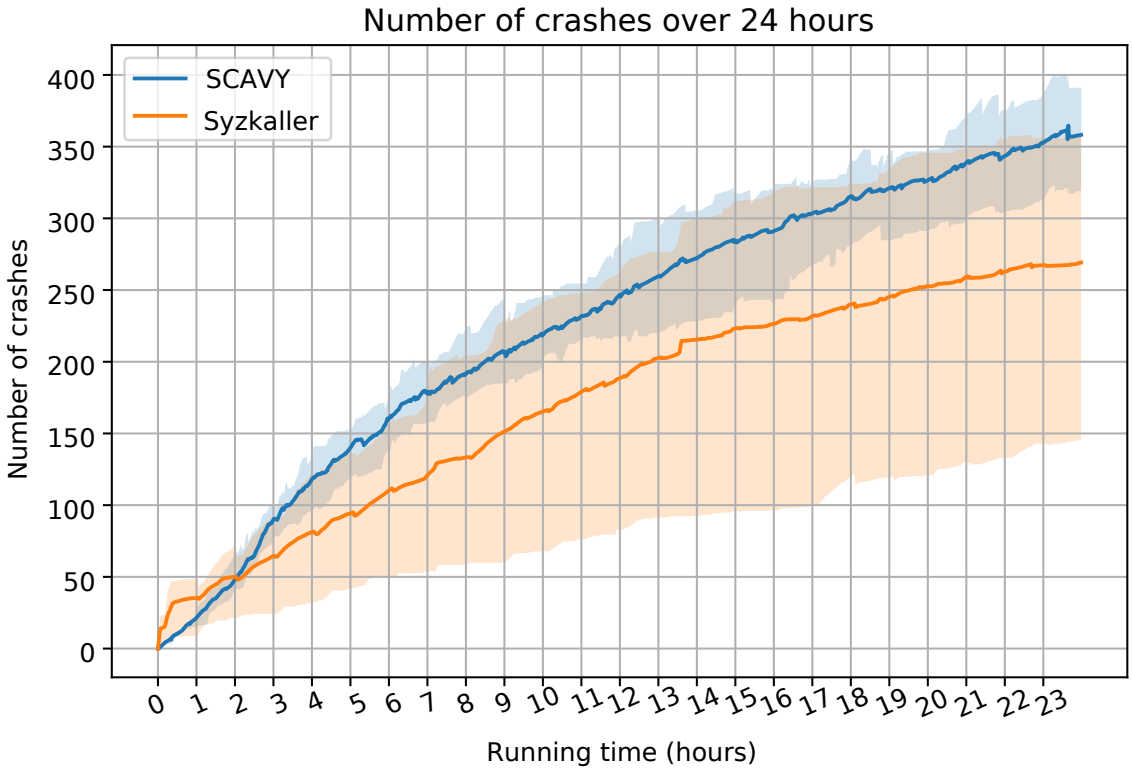

Figure 4.9: # of Crashes Observed

code coverage is irrelevant to finding memory targets.

4.7.3 Exploitable State Detection

To understand the impact of our new definition of exploitable state in finding memory targets, we conduct two evaluations. First, we compare the scope of our exploitable state definition to the commonly studied ones, such as write-what-where and control flow hijacking. Second, we demonstrate the efficacy of SCAVY by presenting its ability to discover both previously known and new memory targets. To conduct this experiment, we ran the fuzzer for 7 days. We identified 2,811 unique fields from 139 distinct kernel structures that could be exploited. The fuzzer

generated a total of 3,863 PoCs whose corresponding object corruptions crashed the kernel.

4.7.3.1 Evaluation of the Exploitable State Definition

We run FUZE's symbolic executor and SCAVY to detect the exploitable states for all the PoC. To avoid the FUZE being stuck in an infinite loop, we disabled around 700K function call sites corresponding to KCOV, Ftrace, our custom driver, and part of KASAN. For a fair comparison, we used a 5-minute timeout (FUZE uses the same timeout). We then ran the same PoCs through our privilege escalation detector step.

FUZE found 354 PoCs that lead to at least one exploitable state, meaning that during the unconstrained symbolic execution, more than one value satisfies the branches taken to reach an exploitable state. The exploitable states FUZE found are mostly attacker-controlled write with a symbolic destination address. We show the number of PoCs that FUZE detected to reach at least 1 exploitable state in Table 4.2 below.

Structure	Field	# PoCs
proc_inode	pid	52
anon_vma	root	36
anon_vma	parent	41
socket_alloc	vfs_inode.i_flags	2
socket_alloc	vfs_inode.xa_head	6
pde_opener	c	20
io_uring_probe	ops	6
file	f_ra	5
io_wq	worker_done.pprev	1
sk_security_struct	peer_sid	7
others	-	178

Table 4.2: **Memory Targets Found by FUZE's Symbolic Execution.** Multiple PoCs for each memory target are detected.

Meanwhile, the detection step of SCAVY flagged *955* PoCs for having a deviation that can lead to an exploitable state. While our definition of exploitable state does not include control flow hijacking and write-what-where (i.e., it is not a superset of the prior work), the additional PoCs SCAVY found may imply the effectiveness of SCAVY's broader definition.

Among the 955 PoCs, many are corrupting the same field. Therefore, we decided to prioritize our manual analysis of the fields with the most positive detections

Structure::field	Prerequisites	Post.	CVE
file::f_mapping	RW, C_S, 216, 8, p	RW, ($\mathbb{F}$)	✷♠♣
✪ task_struct::cred	RW, C_S, 1712, 8, p	X, ($\mathbb{S}$)	♦
task_struct::mm	RW, C_S, 1080, 8, p	R, ($\mathbb{M}_p$)	♦
task_struct::active_mm	RW, C_S, 1088, 8, p	R, ($\mathbb{M}_p$)	♦
task_struct::vma_cache	RW, C_S, 1104, 8, p	R, ($\mathbb{M}_p$)	♦
address_space::i_pages	RW, C_G, 8, 8, p	W, ($\mathbb{F}$)	✷
vm_area_struct::vm_file	RW, C_S, 160, 8, p	RW, ($\mathbb{M}_v$)	✷♠♥
inode::i_uid	W, C_S, 4, 4, 0	W, ($\mathbb{F}$)	♣♠
inode::i_mapping	RW, C_S, 4, 4, p	RW, ($\mathbb{F}$)	♣♠
inode::i_pipe	RW, C_S, 568, 8, p	R, ($\mathbb{M}$)	♣♠
pipe_inode_info::bufs	W, C_G, 152, 8, p	R, ($\mathbb{M}_v$)	♦
kioctx::aio_ring_file	RW, C_S, 512, 8, p	R, ($\mathbb{M}_v$)	✷
kioctx::internal_pages	RW, C_S, 448, 8, p	R, ($\mathbb{M}_v$)	♣
aio_kiocb::ki_filp	RW, C_S, 0, 8, p	RW, ($\mathbb{M}_v$)	—
key::user	RW, C_S, 72, 8, p	RW, ($\mathbb{M}_v$)	✷
key::description	W, C_S, 32, 8, p	R, ($\mathbb{M}_k$)	✣✷
key::perm	W, C_S, 112, 8, 3¡¡25	R, ($\mathbb{M}_v$)	✷
shmem_inode_info::i_mapping	RW, C_S, 168, 8, p	R, ($\mathbb{M}_v$)	✷
✪ cred::cap_bset	W, C_S, 64, 8, 2^{21}	R, ($\mathbb{M}_v$)	♦
✪ cred::euid	W, C_S, 20, 4, 0	X, ($\mathbb{S}$)	♦

Table 4.3: **Corruptions Found to Lead to Exploitable States.** Prerequisites are of the form: <exploit capabilities, generic/special slab cache,offset,size, value (kernel pointer or specific value)>. Post-conditions are of the form <observed capability, resource>. Resources are: Files($\mathbb{F}$), Syscall ($\mathbb{S}$) and Memories of Process($\mathbb{M}_p$), Kernel($\mathbb{M}_k$) and other Virtual($\mathbb{M}_v$). ✪ indicates known field. We evaluate the exploitability of the corruptions using real-world CVEs marked in the table as: CVE-2010-2959 (♣), CVE-2014-3153 (♦), CVE-2016-0728 (✣), CVE-2017-7184 (♥), CVE-2017-7308 (♠), CVE-2022-27666 (✷).

and the ones we found within our expertise to analyze. In Table 4.3, we show 20 fields that were manually verified to lead to privileged operations and the exploitable state that was detected based on our definition in section 4.1. The rest of the fields are presented in Table 4.4 below. It is worth noting that the corruptions we report in this paper are *a subset of PoCs* we detected (in total, we detect 68).

Structure::Field (offset)	Structure::Field (offset)
anon_vma::parent::rb_root (+0)	mount::mnt_mounts::prev (+0)
anon_vma::refcount (+0)	pde_opener::file (+0)
anon_vma::root::count (+0)	pid_namespace::ucounts (+0)
anon_vma::rwsem (+0)	pipe_buffer::ops (+0)
anon_vma::rwsem (+24)	proc_inode::pid (+0)
avc_node::ae::pprev (+16)	proc_inode::pid (+4)
bio::bi_private (+0)	proc_inode::sibling_inodes (+8)
ctl_table_header::(anon) (+16)	proc_inode::sysctl (+0)
dentry::d_lockref (+0)	proc_inode::vfs_inode::i_acl (+0)
dentry::d_op (+0)	proc_inode::vfs_inode::i_data (+0)
dnotify_mark::fsn_mark (+56)	proc_inode::vfs_inode::i_data (+8)
ext4_inode_info::vfs_inode (+280)	proc_maps_private::mm (+0)
file_lock::fl_link::prev (+0)	shmem_inode_info::swaplist (+0)
file_lock::fl_list::pprev (+0)	vfs_inode::i_data (+0)
file_lock::fl_wait (+8)	vfs_inode::i_lock (+0)
files_struct::fdtab (+64)	vfs_inode::i_mapping (+0)
fs_struct::pwd::dentry (+0)	sock::sk_callback_lock (+0)
fs_struct::seq (+0)	sock::sk_peer_cred (+0)
fs_struct::users (+0)	socket::i_acl (+0)
hugetlbfs_inode_info::vfs_inode (+272)	socket::i_default_acl (+0)
hugetlbfs_inode_info::vfs_inode (+60)	socket::i_op (+0)
inode::i_data::prev (+0)	vfs_inode::i_data::prev (+0)
inode::i_lock (+0)	vfs_inode::i_lru::next (+0)
journal_head::b_triggers (+0)	vfs_inode::i_sb_list::next (+0)
kioctx::users (+0)	user_struct::epoll_watches (+32)

Table 4.4: Potential Privilege Escalations Detected by SCAVY but *Unverified*.

In subsubsection 4.7.3.2, we present details of the manual analysis we performed on the deviations detected by SCAVY. In the second column of Table 4.3, we summarize the capabilities required for the attacker to correctly corrupt the field, such as requiring both read and write or just write capability. For example, to exploit using `key::description`, the attacker needs to have the capability to **W**rite **8** bytes into the `key` cache at offset **32** the object with the value **0**. In the third column, we summarize the post-conditions, essentially escalated privileges, such as reading a pipe or writing a file. For example, after corrupting `key::description` with a valid kernel pointer, the attacker can read arbitrary kernel memory.

Evaluating with Real world Vulnerabilities. In the last column, we evaluate the reachability of SCAVY found memory targets using real-world exploits. Specifically, we first obtain exploits from KHeaps [84], as the authors provide vulnerable kernel environment that can reproduce their exploits. For each exploit, we first identify the part of the exploit that creates (or sprays) the victim kernel structure. Then, we check the exploit's memory corruption capability with the SCAVY's memory target's requirements. If the exploit has sufficient capability, we massage the memory layout [83] for our victim kernel structure (i.e., SCAVY's memory target) to be within the corruption range. Next, we replace the victim kernel structure creation (or spraying) code with SCAVY's PoC's code to allocate SCAVY's memory target. Finally, we trigger the vulnerability to corrupt SCAVY's memory target with an arbitrary value. After that, we replace the part accessing the corrupted memory in the original exploit with SCAVY's PoC's privilege escalation checking code, which invokes privilege-assessing system calls that eventually access corrupted

memory and raise crashes if corrupted. If we get a crash with the corrupted value in its panic dump, we mark column 4 of this field with the respective CVE ID. We publish the modified exploits that can corrupt SCAVY targets in our repository.

4.7.3.2 Analysis of Detected Exploitable States

We present our manual analysis for the proof-of-concept code detected by SCAVY, focusing on exploring concrete privilege escalation exploits from the detected exploitable states.

Case 1: `file::f_mapping` The `file::f_mapping` memory target can be leveraged to create exploits for two different scenarios.

Scenario 1: Attacking Inaccessible File. An exploit creates a root-owned process opening a privileged file, which is completely inaccessible in an unprivileged process. For example, it can spawn many '`passwd`' processes, spraying the slab memory with the '`/etc/shadow`' file's `file` kernel structures. Then, the exploit also creates an unprivileged file such as a temp file (e.g., `/tmp/file`). Next, the exploit leverages a vulnerability with a read capability to read the `file::f_mapping` of the '`/etc/shadow`,' which will be copied to the `file::f_mapping` of the temp file through a vulnerability with a write capability. In other words, the values of `file::f_mapping` of `/etc/shadow` and the temp file are swapped. After this, reading and writing the temp file is accessing the `/etc/shadow`'s content.

Scenario 2: Attacking Unwritable (but readable) File. Scenario 1 creates many root-owned processes (e.g., `passwd` processes), which might raise suspicion. If

one wants to write an unwritable file (instead of read and write an inaccessible file, as we see in scenario 1), one can achieve it more subtly. Specifically, an exploit can open many '`/etc/passwd`' files in read mode and a temp file in read/write mode. Swapping the `file::f_mapping` of the '`/etc/passwd`' and the temp file allows the exploit to write the '`/etc/passwd`', escalating the privilege. With the privilege, it can create a root-level account by editing the '`/etc/passwd`' file.

Case 2: `vm_area_struct::vm_file`

This memory target can, in practice, be used to access privileged files' contents if they exist in shared memory-mapped files (i.e., mapped with `mmap()` with the MAP_SHARED flag). Specifically, an exploit first creates a shared memory of a temp file with read and write permissions, which will create a `vm_area_struct::vm_file` kernel structure. Then, it overwrites the `vm_area_struct::vm_file` of the temp file with the values of the `vm_area_struct::vm_file` of a root-owned file, using vulnerabilities with read and write capabilities. After the corruption, an exploit invokes `msync` to synchronize the mapping with the corrupted new `vm_area_struct::vm_file`.

Note that, in subsection 4.2.1, we show that one can modify our exploit of `CVE-2022-27666` to write into any of the dummy file's `vm_area_struct` so that it would access the content of `/etc/passwd`.

Listing 4.3: Syzkaller representation of the PoC.

```
#23:08:35 executing program 0 (corruption `
#   %struct.anon_vma_chain*+40 (8)` at call 4):
r0 = semget(0x3, 0x3, 0x481)
```

```
getgroups(0x2, &(0x7f0000000180)=[<r1=>0xee01,<r2=>0xee01])
ioctl$NS_GET_OWNER_UID(0xffffffffffffffff, 0xb704,
    &(0x7f0000000200)=<r3=>0x0)
semctl$IPC_SET(r0, 0x0, 0x1, &(0x7f0000000240)=
    {{0x2,0xee01, r2, r3, 0xee01, 0xa0,
    0x5f3}, 0xffffffff,0x0, 0x0, 0x0, 0x0, 0x0, 0x7})
r4 = semget(0x3, 0x4, 0x40)
semctl$GETVAL(r4, 0x2, 0xc, &(0x7f0000000000)=""/248)
semget(0x0, 0x2, 0x4)
getgroups(0x1, &(0x7f0000000100)=[<r5=>r1])
r6 = getegid()
getgroups(0x5, &(0x7f0000000140)=[r5, r2, r2, r6, r5])
```

Case 3: `anon_vma_chain::rb` SCAVY found this kernel data structure field as a potential memory target. However, we did not include it as a memory target as corrupting it does not escalate privilege.

SCAVY discovered this memory target by observing a crash during the execution of `semctl()` after corrupting the `anon_vma_chain::rb` field with a random value. Specifically, it crashes at `semctl$GETVAL` shown in line 8 in Listing 4.3. We further analyze the structure to understand the crash. In particular, the `anon_vma_chain` structure is used for anonymous virtual memory, which is essentially shared memory not backed by a file system. The `rb` field links the `anon_vma_chain` structure into a red-black tree [120] for search optimization. As a result, corrupting it with a random

value, which is not a valid pointer, caused a memory dereference error. While we could not enhance the case to replace the `rb`'s value with another valid `rb`'s value, replacing a valid value may impact the search operations using the red-black tree.

4.7.3.3 Evaluation of the Privilege Escalation Detector

Among the 955 detected behavior deviations, we find that code coverage is the most common deviation. Other significant deviations include accessing random addresses (when modifying data pointer fields) and privilege de-escalations (when modifying non-pointer fields). Table 4.5 summarizes our results. We also analyze the common system calls where the deviations occur. Table 4.6 shows the top 10 system calls exhibit deviations, helping detect reaching exploitable states. Among them, we observe the `seteuid` system call that we additionally instrument, helping 16 cases out of 955 (1.5%). Others are mostly I/O related system calls (e.g., `pread64`, `ioctl`, `socketpair`, and `sendmsg`).

Deviation	**Count**
Coverage	658
Corruption hits	452
Return values	200
Buffers	176

Table 4.5: Count of deviation conditions.

Deviating Syscall	Count
pread64	283
io_submit	35
ioctl$sock_SIOCGIFINDEX_8b2154	27
ioctl$ifreq_SIOCGIFINDEX_team	26
ioctl$sock_SIOCGIFINDEX_8b211	23
seteuid	16
ioctl$BTRFS_IOC_GET_SUBVOL_ROOTREF	12
ioctl$BTRFS_IOC_INO_LOOKUP_USER	11
socketpair$nbd	10
sendmsg$ETHTOOL_MSG_LINKINFO_GET	9

Table 4.6: Count of syscalls demonstrating deviation

4.7.3.4 Exploiting Real Vulnerabilities

We evaluate SCAVY found memory targets in terms of writing exploits with real-world vulnerabilities to illustrate its real-world impact and practicality. Our result is summarized in Table 4.7. Note that the vulnerabilities are from different software weaknesses, as denoted by the CWE (Common Weakness Enumeration), showing that SCAVY's memory targets do not have specific requirements on the type of vulnerabilities.

We obtain the exploits from various sources. The first three exploits (1∼3) are from the original KHeaps [84] paper. The fourth exploit is borrowed from a public PoC[6]. Other exploits are from publicly available PoCs [83, 121]. For all

[6]https://gist.github.com/PerceptionPointTeam/18b1e86d1c0f8531ff8f

	CVE	CWE	Exploited Struct/Field
1	2017-7308	Type Conversion	vm_area_struct::vm_file
2	2017-7184	Input Validation	vm_area_struct::vm_file
3	2010-2959	Int Overflow	kioctx::internal_pages
4	2016-0728	Use-After-Free	key::description
5	2022-27666*	OOB Write	vm_area_struct::vm_file
6	2022-27666*	OOB Write	file::f_mapping
7	2009-3547	nullptr deref.	pipe_inode_info::bufs
8	2014-3153	Input Validation	cred::euid
9	2017-11176	Use-After-Free	task_struct::cred
10	2004-1235	Race condition	file_struct::fops

Table 4.7: Modified Real World Exploits (*) indicates constructed fully functional exploit).

the exploits except for the `2016-0728`'s exploit, we follow the procedure described in subsubsection 4.7.3.1 to modify the exploits to corrupt SCAVY memory targets. The exploit for `2016-0728` is shown in subsection 4.2.2. The exploits `5` and `6` are end-to-end exploits using `2022-27666` with SCAVY found memory targets. The entire exploits are presented in subsection 4.7.4. The exploits `7`∼`9`, we found that the original exploits corrupt kernel structures that contains SCAVY's memory targets (in Table 4.3), while they target a different field to corrupt.

Note that we were unable to reproduce the `2004-1235` exploit as we could not reproduce the vulnerable environment (e.g., the kernel and OS); however, based on the developer email chain [122], we deduced that the exploit could control the content of a `vm_area_struct` and it had the `vm_file` field in the target kernel version (`2.6`), implying that the exploit can corrupt a SCAVY memory target. We release the exploits and vulnerable systems' VMs on [123].

4.7.4 Exploiting CVE-2022-27666

In this section I explain our 2 end-to-end exploits that use the SCAVY found memory targets to achieve privilege escalation. The first exploit uses the `file::f_mapping` memory target, and the second exploit uses the `vm_area_struct::vm_file` memory target.

4.7.4.1 Exploiting using `file::f_mapping`

We select the first primitive identified by SCAVY in Table 4.3. To use `file::f_mapping`, it requires a memory read capability to read a valid `address_space` structure, which is then used to overwrite the `f_mapping`.

Unlike existing exploit [83], our method does not require the breaking of KASLR or the overwriting of *modprobe_path*. As a result, the exploit doesn't require the attacker to have the kernel symbols (which is usually difficult for a custom kernel). Additionally, the system is more stable after the exploit has been executed because the global variable `modprobe_path` is not modified.

The exploit requires memory massaging of Linux page allocator. Hence, we borrow the same noise-mitigation and massaging technique from an existing exploit [83]. The rest of the exploit is summarized in six steps as follows:

1. Allocate `user_key_payload` next to the vulnerable structure. Then, overflow and corrupt `user_key_payload::datalen` to obtain the read out of the bounds (OOB) capability.
2. Open two files: one dummy file with read/write permissions and another root-owned unwritable file that can only be opened read-only by non-root users (e.g., `/etc/passwd`).
3. Repeatedly call `open()`[7] to assure the `file::f_mapping` structures allocate next

[7]There is a limit of 1,000 invocations of `open()`. However, most of the allocations would fall in the memory holes created in step 1, without exhausting the limit. If necessary, one can use the following method to overcome the 1,000 limit: open both files once and `mmap` the opened files until the `vm_area_struct` are allocated next to the corrupted `user_key_payload`.

to user key payload. Leak the two `vm_area_struct::file` pointers of the opened files.

4. Rearrange the memory to now put the `msg_msg` structure next to the vulnerable structure and overflow into it such that the `msg_msg::next` points to the leaked `file` structure. With the corruption, reading `msg_msg` leaks the content of the `file` structure of the `/etc/passwd`.
5. Rearrange the kernel memory to use the `msg_msg` to achieve arbitrary write capability [124], and use it to overwrite the `file::f_mapping` of the dummy file with the leaked `f_mapping` value of the `/etc/passwd`.
6. Call `write()` with the dummy file's descriptor to append a new root-level account on `/etc/passwd`. As shown in the blog post [125], the root privilege can be obtained by simply running `su` with the appended account.

In Figure 4.10, we illustrate the memory layout for the steps 3 to 5. After step 5, as SCAVY detected, the `/etc/passwd` becomes writable through the dummy file's descriptor.

4.7.4.2 Exploiting using `vm_area_struct::vm_file`

This exploit corrupts `vm_area_struct::vm_file` by using `CVE-2022-27666`. Similar to the exploit above, it requires both read and write capabilities to obtain a valid `vm_file` pointer of another `vm_area_struct` and overwrite it to another structure. Also, it does not require bypassing KASLR and corrupting `modprobe_path`.

The exploit shares the first two steps of the exploit above: allocating a page

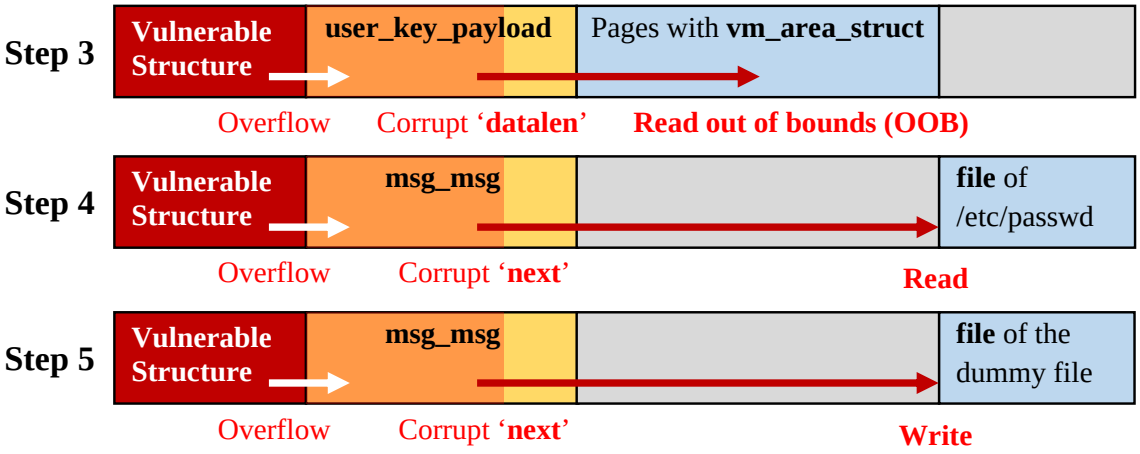

Figure 4.10: Visualizing Exploit of CVE-2022-27666.

of `struct user_key_payload` next to the vulnerable structure and corrupting its `datalen` field to obtain the OOB capability (when `keyctl()` is called with the corrupted structure). Then, it also opens two files: a dummy file with read/write permissions and a root-owned unwritable file. It diverts from the step 3 as shown below:

0. Perform heap feng-shui to create holes in memory to protect our exploit from runtime noise in memory allocations.
1. Allocate a page of `struct user_key_payload` next to the vulnerable structure. Then, overflow and corrupt its `datalen` field, obtaining the out of bounds read capability when `keyctl()` is called with the corrupted structure. This allows the next call to *keyctl* read out of the bounds of the user key payload
2. Open 2 files. One dummy file with read/write permissions and a root-owned unwritable file that can only be read-only by non-root users such as `/etc/passwd`.
3. `mmap()` the opened files until the `vm_area_struct` are allocated next to the corrupted `user_key_payload`. Leak `vm_area_struct::vm_file` of `/etc/passwd`.

4. Rearrange kernel memory to use the `msg_msg` arbitrary write capability [124] to overwrite the `vm_area_struct::vm_file` pointer of the dummy file with the leaked `vm_file`'s value in the previous step.
5. After step 4, any memory access of the mapped memory of the dummy file becomes accessing `/etc/passwd`, allowing arbitrary read and write to the file. A root privilege can be obtained after adding a line with a new root account in the file and then running the `su` command with the added credential. Details can be found on [125].

By the end of step 5, the `/etc/passwd` file becomes writable through the dummy file's descriptor. This allows the attacker to append a new root-level account to the `/etc/passwd` file, similar to the exploit above.

4.7.5 Exploiting real world vulnerabilities

The exploits below are examples we found from public exploits that define the structures SCAVY detected to contain fields that can be corrupted to perform privilege escalation. To reproduce them we setup their respective vulnerable systems, fix and compile the original exploit and verify that it works. Then we modify the part where they corrupt SCAVY's struct to instead corrupt the field SCAVY found. At last, we insert code from stage 3 of SCAVY's PoC to cause access to SCAVY's field and crash.

4.7.5.1 CVE-2009-3547

We built Ubuntu 09.10 with the required compiler. We set /proc/sys/vm/mmap_min_addr to 0, allowing a process to map the null page. The bug, a null pointer dereference allows an attacker to trick the kernel into accessing the user-mapped null page and it's content. The content in that page is a pipe_inode_info structure. The original exploit makes a chain of fake structures ending in the attacker creating a pointer table structure, ultimately causing the kernel to execute from the attacker-controlled locations. We modify the pipe_inode_info::buf instead to a random value and call read() on the pipe, causing it to crash in the same way as the PoC SCAVY generated.

4.7.5.2 CVE-2014-3153

We compiled and verified the exploit for CentOS 7.0.1406. Originally, the bug provides a memory corruption capability on rt_mutex_waiter structure, allowing the attacker to corrupt the thread_info::addr_limit, a widely used technique [3] allowing full RW capability to arbitrary kernel memory. From there the exploit finds and overwrites its own task_struct::cred::uid, effectively become the root user. Once the task_struct is found we modify the exploit to corrupt task_struct::mm instead with 0xdeadc0de and triggered it by simply calling mmap. We observed the kernel panic when trying to dereference the value above, same as SCAVY's PoC did.

4.7.5.3 `CVE-2017-11176`

We compiled and verified the exploit for `Debian 8.6.0`. Originally, the exploit has the capability to overwrite `__wait_queue::wait_queue_func_t` [126] causing the kernel to execute exploit's payload. In that payload the exploit modifies `thread_info::addr_limit` effectively reaching the same capability as subsubsection 4.7.5.2. We modify the payload to access the `task_struct::cred` instead and crash on program termination due to invalid `cred` pointer.

4.8 Discussion and future work

SCAVY Facilitating Defenses. Even though SCAVY is mostly focused on facilitating the exploitation stage we believe a defender can benefit from it too. SCAVY exposes sensitive fields that lead to privilege escalation. With this information a defender doesn't need to rely on the post-attack analysis to figure out how to defend the sensitive fields. For example, defenses for kernel data structure fields, such as XOR'ing their values, redzoning the fields [127], and applying write-once memory [128], while they cause significant overhead if applied to every kernel structure [6]. Kernel protections such as KASAN slab redzoning are usually disabled in production due to the intolerable overhead. We believe that instrumenting access to sensitive fields such that redzones are check only on those can lead to lower overhead, making it possible to deploy in production. The object tracking capability of SCAVY may be used in existing kernel benchmarks to give the analyst a better idea on what objects are commonly being allocated and what fields are accessed more

often. This can lead a defender to deploying a more efficient defense against memory corruptions. Another possible defense is the memory isolation proposed by Dirty-Cred [5]. This defense though doesn't work against some memory overwrites such as the *uid* overwrite with zeros. In this case the defender may add a new field to the structure's last 8 bytes and fill it with random data, then use this data to XOR the sensitive fields in that structure. This can lead to a memory overhead depending on how often the structure gets allocated. The defender may also move the sensitive fields in the middle or the end of the object, surrounding them with pointer fields to make the overflows less stable, because now the attacker will be overwriting other pointers with illegitimate data causing the kernel to panic on dereference. A promising idea is that of protectable memory, proposed in 2018 [128]. The premise is to add a new function, `pmalloc`, to the allocator which makes the sensitive data harder to find and make them read only once they are populated. The authors propose the "rare write" mechanism to make the data writable only during a brief window frame. One problem they highlight is the lack of users. We hope that SCAVY can highlight some potential users and bring this idea to production.

Limitations. There exist a few sources of false positive in SCAVY's analysis. First, since SCAVY considers *any* differences between the executions with and without memory corruption as a potential privilege escalation if a divergent is caused by other reasons, such as a corrupted semaphore or lock fields causing an execution to hang and timeout, it would lead to a false positive. Second, if the device-specific pointers are corrupted, an execution may divert and be detected by SCAVY without escalating privilege, meaning it is a false positive case. For example, `bio::bi_private` is

dedicated to device-specific structs and is usually `null`. When releasing, the kernel frees the pointer if it is not null, causing a deviation in code coverage. While SCAVY detects it, as per our definition, it does not lead to an exploitable state. Hence, it is considered a false positive. However, it is still a valid exploitation (while not a privilege escalation) as an attacker can corrupt this field to get an arbitrary-free capability. The `refcount` fields are also considered false positives as they do not escalate privilege but they can lead to valid use-after-free capabilities. Third, SCAVY may produce a false detection when modifying the read head of a file. We didn't observe it, but it's theoretically possible because this may affect the return buffer of a `read` syscall. The detection would be an FP, because the attacker doesn't get any privileged read capabilities. At last, we rely on a differential analysis method to detect privilege changes, meaning that SCAVY can only detect when system calls have *visible changes from userland* via the system calls in Table 4.1. For example, corrupting `addr_limit` to a larger value would not have immediately observable execution divergence, which will be missed by SCAVY.

Finding kernel object's data type. Exploitation aside, SCAVY proposes a novel data type tracking mechanism and incorporates it with Syzkaller, introducing another dimension in the coverage guided fuzzing. Our methodology can be used instead of static analysis to have a precise data type of the kernel objects. Additionally, since Syzkaller has a module that attempts to recreate the PoC that caused the crash it's important to know all objects allocated during the runtime since in many cases the heap layout might differ. SCAVY can track all object allocations from all the running processes, further assisting an analyst in reproducing the crash

in case the automated Syzkaller fails to do so. At last, our methodology can be used to track the data type of the objects that are being allocated when analyzing a kernel exploit.

Future work. Our work only exposes the impact of single-field corruptions, in reality an attacker usually has memory corruption capability of more than a single field, especially in use-after-frees when they can use the well known `msg_msg` [124] method to completely control all the data in the overlapping object. SCAVY also doesn't consider race condition bugs. An attacker may be able to exploit a corruption similar to the credential swapping method proposed by DirtyCred [5]. We leave such analysis as future work. It requires that the fuzzer introduces the corruption while a syscall is being executed. This imposes more challenges in terms of controlling the time window and reproducibility of the PoC. Another shortcoming of the way we search for sensitive fields is the detection of out-of-bound read privileges. Coined with the term elastic objects [81], a corruption in these structures allows the attacker to read out of bounds into kernel memory. This is undetectable by our fuzzer since the fuzzer doesn't rely on the corruption to issue an invalid system call with the expectation that it might work. Additionally, we mentioned throughout the paper that SCAVY may help existing bug exploration tools in reaching an exploitable state for more PoCs. We leave the implementation of a joins system as future work.

Responsible Disclosure. We have responsibly informed and shared all the findings and artifacts (e.g., exploits) with the Linux maintainers and discussed its potential security consequences. We have their approval to disclose SCAVY, our findings as well as the exploits that we wrote.

Chapter 5: Conclusion

In my thesis I focus on analyzing and exploring attackers' behavior in the wild. I analyze the behavior of malware, PUP and benign samples in the wild, using execution traces collected from 5.4M real hosts from around the world. I show that malware exhibits different levels of variability in different actions and evaluate prior state-of-the-art detectors and classifiers on the dataset to highlight the issues with using sandbox data in evaluations. I leverage an observation from the first project, where variability in the malware samples happens due to running privilege escalation exploits.

For these reasons, I propose a new methodology to systematically discover sensitive memory corruption targets that cause privilege escalation, further challenging the definition of exploitability from prior works. I explore a new method of finding severe memory corruption vulnerabilities independent of any particular vulnerability. I designed and implemented SCAVY, which efficiently searches for memory targets that can lead to privilege escalation in the Linux kernel. Unlike prior work, I introduce a complete definition of a capability within the weird state machine, allowing for a systematic exploration of the search space instead of simply relying on the fuzzer to randomly reach the final exploitable state.

SCAVY can be used in accordance with existing bug-exploration tools to better assess the severity of a bug. Additionally, this framework can be used by threat actors to speed up their attempts at writing a full end-to-end exploit. In the end they will only need to formally write their initial capability (as provided by the bug they have found) and the system will automatically suggest a multiple exploit chains. SCAVY discovered 17 memory targets in 12 kernel structures that can lead the system into an exploitable state when corrupted. At last, I show the impact of SCAVY-found memory targets by demonstrating real-world PoCs of 9 CVEs corrupting the 17 memory targets.

Bibliography

[1] Dan Rosenberg. Interesting kernel exploit posted, 2010. https://lwn.net/Articles/419141/.

[2] Maddie Stone. CVE-2019-2215: Android use-after-free in Binder, 2020. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[3] Mark Brand. In-the-Wild Series: Android Exploits, 2021. https://googleprojectzero.blogspot.com/2021/01/in-wild-series-android-exploits.html.

[4] Mark Rutland and Catalin Marinas. arm64: uaccess: remove set_fs(), 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3d2403fd10a1dbb359b154af41ffed9f2a7520e8.

[5] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1963–1976, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. Additional kernel observer: Privilege escalation attack prevention mechanism focusing on system call privilege changes. *Int. J. Inf. Secur.*, 20(4):461–473, aug 2021.

[7] Kees Cook. security things in linux v4.14, Nov 2017. https://outflux.net/blog/archives/2017/11/14/security-things-in-linux-v4-14/.

[8] Thomas Garnier. mm: Slab freelist randomization, Apr 2016. https://lwn.net/Articles/685047/.

[9] Jonathan Corbet. Supervisor mode access prevention, Sep 2012. https://lwn.net/Articles/517475/.

[10] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194. IEEE, 2016.

[11] Kees Cook. Add slub free list pointer obfuscation, 2017.

[12] Ruiqi GONG. Randomized slab caches for kmalloc, 2023.

[13] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[14] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 780–797, USA, 2018. USENIX Association.

[15] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2078–2095. IEEE, 2022.

[16] Corjail: From null byte overflow to docker escape exploiting `poll˙list` objects in the linux kernel, 2023.

[17] Vitaly Nikolenko. Cve-2016-6187: Exploiting linux kernel heap off-by-one, 2023.

[18] Andy Nguyen. Cve-2021-22555: Turning `“x00“x00` into `10000$`, 2023.

[19] Andrey Konovalov. Cve-2017-1000112: Exploiting an out-of-bounds bug in the linux kernel ufo packets, 2023.

[20] Michał Praszmo. Ramnit – in-depth analysis. https://www.cert.pl/en/news/single/ramnit-in-depth-analysis/.

[21] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 65–79. IEEE, may 2012.

[22] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium 2019)*, pages 729–746, 2019.

[23] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and Efficient Malware Detection at the End Host. In *Presented as part of the 18th USENIX Security Symposium (USENIX Security 09)*, Montreal, Canada, 2009. USENIX.

[24] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 5, New York, New York, USA, 2007. ACM Press.

[25] Platon Kotzias, Leyla Bilge, Pierre-Antoine Vervier, and Juan Caballero. Mind Your Own Business: A Longitudinal Study of Threats and Vulnerabilities in Enterprises. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[26] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, page 122, New York, New York, USA, 2012. ACM Press.

[27] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C Mitchell. A layered architecture for detecting malicious behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 78–97. Springer, 2008.

[28] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *2010 IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2010.

[29] Dhilung Kirat and Giovanni Vigna. MalGene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the ACM Conference on Computer and Communications Security*, volume 2015-Octob, pages 769–780, New York, New York, USA, 2015. ACM Press.

[30] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. *NDSS*, apr 2010.

[31] Ulrich Bayer, P.M. Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[32] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the

mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[33] Hiroshi Konno, Hiroshi Shirakawa, and Hiroaki Yamazaki. A mean-absolute deviation-skewness portfolio optimization model. *Annals of Operations Research*, 45(1):205–220, 1993.

[34] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[35] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6961 LNCS, pages 338–357. Springer, Berlin, Heidelberg, 2011.

[36] Yara. https://virustotal.github.io/yara/.

[37] Florian Roth. Generic Signature Format for SIEM Systems. https://github.com/Neo23x0/sigma.

[38] Philipp Trinius, Carsten Willems, Thorsten Holz, and Konrad Rieck. A Malware Instruction Set for Behavior-Based Analysis. *Sicherheit Schutz und Zuverlässigkeit SICHERHEIT*, 2011.

[39] ss64. Quotes, Escape Characters, Delimiters - Windows CMD - SS64.com. https://ss64.com/nt/syntax-esc.html.

[40] Marcos Sebastian, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses*, 2016.

[41] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[42] TrendMicro. TrojanSpy.Win32.GLUPTEBA.A - Threat Encyclopedia - Trend Micro USA. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/trojanspy.win32.glupteba.a.

[43] Z Cliffe Schreuders, Thomas Shaw, Mohammad Shan-A-Khuda, Gajendra Ravichandran, Jason Keighley, and Mihai Ordean. Security scenario generator (secgen): A framework for generating randomly vulnerable rich-scenario vms for learning computer security and hosting {CTF} events. In *2017 {USENIX} Workshop on Advances in Security Education ({ASE} 17)*, 2017.

[44] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024. IEEE, 2017.

[45] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. *Recent Advances in Intrusion Detection*, pages 178–197, 2007.

[46] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: Using system-centric models for malware protection. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 399–412, New York, NY, USA, 2010. Association for Computing Machinery.

[47] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 231–245. IEEE, may 2007.

[48] Jedidiah R Crandall, Gary Wassermann, Daniela AS de Oliveira, Zhendong Su, S Felix Wu, and Frederic T Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. *ACM SIGOPS Operating Systems Review*, 40(5):25–36, 2006.

[49] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 732–744, 2015.

[50] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)*, pages 161–175. IEEE, 2018.

[51] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.

[52] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, jun 2011.

[53] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[54] Microsoft resumes security updates with 'largest' patch tuesday release. Redmont Mag, 30 March 2017. https://redmondmag.com/articles/2017/03/14/march-2017-security-updates.aspx.

[55] Massive microsoft patch tuesday security update for march. Qualys, 30 March 2017. https://blog.qualys.com/laws-of-vulnerabilities/2017/03/14/massive-security-update-from-microsoft-for-march.

[56] Thomas F Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[57] Linux. Android application sandbox, 2023.

[58] Erin Avllazagaj, Ziyun Zhu, Leyla Bilge, Davide Balzarotti, and Tudor Dumitraș. When malware changed its mind: An empirical study of variable program behaviors in the real world. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3487–3504, 2021.

[59] Apache web server, 2023.

[60] mitre. Cwe-548: Exposure of information through directory listing, 2023.

[61] NVD. Cve-2022-27666 detail, 2023.

[62] Collin Mulliner. Cve-2016-0728 vs android, 2023.

[63] Michael Mimoso. Serious linux kernel vulnerability patched, 2023.

[64] Syzkaller. https://github.com/google/syzkaller. Accessed: 2023-03-20.

[65] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[66] Trinity. https://github.com/kernelslacker/trinity. Accessed: 2023-03-20.

[67] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2345–2358, New York, NY, USA, 2017. Association for Computing Machinery.

[68] FSecureLABS. Kernelfuzzer: Cross platform kernel fuzzer framework, 2023.

[69] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, Boston, MA, August 2022. USENIX Association.

[70] The kernel address sanitizer (kasan), 2023.

[71] The kernel memory sanitizer (kmsan), 2023.

[72] Kernel thread sanitizer (ktsan), 2023.

[73] The kernel concurrency sanitizer (kcsan), 2023.

[74] Google. Linux kernel sanitizers, 2023.

[75] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. Alphaexp: An expert system for identifying security-sensitive kernel objects. In *USENIX Security*, August 2023.

[76] Yueqi Chen and Xinyu Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1707–1722, New York, NY, USA, 2019. Association for Computing Machinery.

[77] Peass - privilege escalation awesome scripts suite. https://github.com/carlospolop/PEASS-ng. Accessed: 2023-03-20.

[78] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In *USENIX Security*, August 2021.

[79] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. Rid: Finding reference count bugs with inconsistent path pair checking. *SIGPLAN Not.*, 51(4):531–544, mar 2016.

[80] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. Linkrid: Vetting imbalance reference counting in linux kernel with symbolic execution. In *USENIX Security*, Boston, MA, aug 2022.

[81] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. CCS '20, New York, NY, USA, 2020. ACM.

[82] Danjun Liu, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, Zhenhao Luo, Tai Yue, and Baosheng Wang. From release to rebirth: Exploiting thanos objects in linux kernel. *IEEE Transactions on Information Forensics and Security*, 18:533–548, 2023.

[83] Xiaochen Zou and Zhiyun Qian. Cve-2022-27666: Exploit esp6 modules in linux kernel, 2022.

[84] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, Boston, MA, August 2022. USENIX Association.

[85] Vitaly Nikolenko. Linux kernel universal heap spray, 2018.

[86] Jake Edge. Kernel address space layout randomization, Oct 2013. https://lwn.net/Articles/569635/.

[87] Vitaly Nikolenko. Practical SMEP bypass techniques on Linux, 2023. http://bit.ly/3GVjvEn.

[88] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. Kepler: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1187–1204, USA, 2019. USENIX Association.

[89] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. Association for Computing Machinery.

[90] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, page 191–205, USA, 2013. IEEE Computer Society.

[91] Meltdown reloaded: Breaking windows kaslr by leaking kva shadow mappings. http://bit.ly/3GVjvEn. Accessed: 2023-03-20.

[92] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. Ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 957–972, USA, 2014. USENIX Association.

[93] linxz. Supervisor mode execution prevention, 2023.

[94] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. Association for Computing Machinery.

[95] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 247–260, New York, NY, USA, 2008. Association for Computing Machinery.

[96] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 1–16, USA, 2017. USENIX Association.

[97] Sparse. https://www.kernel.org/doc/html/latest/dev-tools/sparse.html. Accessed: 2023-03-20.

[98] Smatch. https://lwn.net/Articles/691882/. Accessed: 2023-03-20.

[99] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678, 2018.

[100] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913, 2018.

[101] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr. checker: A soundy analysis for linux kernel drivers. In *USENIX Security Symposium*, pages 1007–1024, 2017.

[102] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with kint. In *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[103] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 920–932. Association for Computing Machinery, 2016.

[104] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. Association for Computing Machinery.

[105] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. Meca: An extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 321–334, New York, NY, USA, 2003. Association for Computing Machinery.

[106] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 363–378, USA, 2016. USENIX Association.

[107] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[108] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In

Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pages 2095–2108, 2018.

[109] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[110] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, Boston, MA, August 2022. USENIX Association.

[111] Linux. Linux kernel coding style, 2022.

[112] llvm. llvm::castinst class reference, 2023.

[113] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel probes, 2023.

[114] Goldwyn Rodrigues. Poke-a-hole and friends, 2009.

[115] Prasad Krishnan. Hardware breakpoint (or watchpoint) usage in linux kernel. In *Proceedings of the Linux Symposium*, pages 149–158. Citeseer, 2009.

[116] addr2line(1) — linux manual page, 2023.

[117] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[118] scikit learn. Jaccard similarity, 2023.

[119] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. CCS, New York, NY, USA, 2018.

[120] Rob Landley, 2007.

[121] SecWiki. linux-kernel-exploits, 2023.

[122] Paul Starzetz. Linux kernel `uselib()` privilege elevation, 2005.

[123] Scavy github repo.

[124] Will, 2021. https://www.willsroot.io/2021/08/corctf-2021-fire-of-salvation-writeup.html.

[125] Cve-2022-27666: My file your memory. https://albocoder.github.io/exploit/2023/03/13/KernelFileExploit.html. Accessed: 2023-03-29.

[126] Nicolas Fabretti, 2018.

[127] Larry H. Linux Kernel Heap Tampering Detection, 2009. http://phrack.org/issues/66/15.html.

[128] Jonathan Corbet. The slab and protected-memory allocators, 2018. https://lwn.net/Articles/753154/.