University of California
Santa Barbara

Finding Attacks and Vulnerabilities in Critical Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Dipanjan Das

Committee in charge:

Professor Giovanni Vigna, Co-Chair
Professor Christopher Kruegel, Co-Chair
Professor Yu Feng

June 2023

The Dissertation of Dipanjan Das is approved.

Professor Yu Feng

Professor Christopher Kruegel, Committee Co-Chair

Professor Giovanni Vigna, Committee Co-Chair

May 2023

Finding Attacks and Vulnerabilities in Critical Systems

by

Dipanjan Das

Acknowledgements

Ph.D is not a solo journey, rather a collective effort. It is a journey to discover the unknown, where you are always at the helm, with some people in front of you, some behind you, and some beside you. People in front are your advisors, people behind you are the friends and family. One beside you is likely to be someone special. I was fortunate enough to have all three, which together made an excellent support system that helped me sail through this journey.

First and foremost, I'd like to thank my Masters advisor prof. Chandrasekaran PanduRangan for igniting a young mind with a pursuit of research in its early days. I am grateful to my Ph.D. advisors, prof. Giovanni Vigna and Christopher Kruegel, for providing necessary criticism, guidance, encouragement, and financial assistance to support my research. Without them, this thesis would not exist. I am grateful to prof. Yu Feng, Kangjie Lu, and Mathias Payer for all those hours of insightful discussion and brainstorming. I cannot forget Noah Spahn, Tim Robinson, Karen Van Gool, and other CS dept. staffs who shielded me from all the red tapes, thus letting me focus on my work.

My loving family, my parents, my uncles and aunts, my cousins—Mousumi, Diptendu, Diptam, and Ankit, my sisters Trisha, Nafisa, and Deepsha, and my dear friends—Sukrit, Shayonee, and Subhabrata—all of them have been excellent sources of positivity, and inspiration that held me up when I felt things were about to fall apart. I am indebted to my fellow researchers at the SecLab, both from the past and the present, whose intellectual discussions, collaborations, and support have enriched my research experience.

Finally, I do not have words to thank Priyanka, my partner and my best friend, for her endless encouragement, belief in my potential, and most importantly, the sacrifices she made throughout. Her very presence made my Ph.D. experience truly unique in many ways.

Curriculum Vitæ

Dipanjan Das

Education

2016 – 2023	Ph.D. in Computer Science University of California, Santa Barbara
2013 – 2015	M.Tech. in Computer Engineering Indian Institute of Technology (IIT), Madras
2006 – 2010	B.Tech. in Computer Engineering Institute of Engineering & Management (IEM), Kolkata

Experience

03/2020 – 06/2020	Research Intern University of Minnesota, Minneapolis
06/2017 – 09/2017	Interim Engineering Intern Qualcomm Technologies, Inc, San Diego
07/2015 – 06/2016	Post-Graduate Research Intern National University of Singapore, Singapore
06/2015 – 07/2015	Software Developer BrowserStack, Mumbai, India
04/2012 – 07/2013	Scientist Engineer Indian Space Research Organization (ISRO), Trivandrum, India
09/2010 – 04/2012	Assistant Systems Engineer Tata Consultancy Services (TCS), Kolkata, India

Publications

1. M. Fleischer*, **D. Das***, P. Bose, W. Bai, C. Kruegel, G. Vigna, and K. Lu, Actor: Action-aware Kernel Fuzzing, In Proceedings of the USENIX Security Symposium (Usenix), 2023.
2. F. Gritti, N. Ruaro, R. McLaughlin, P. Bose, **D. Das**, I. Grishchenko, C. Kruegel, and G. Vigna, Confusum Contractum: Confused Deputy Vulnerabilities in Ethereum Smart Contracts, In Proceedings of the USENIX Security Symposium (Usenix), 2023.
3. P. Bose, **D. Das**, S. Vasan, I. Grishchenko, A. Continella, A. Bianchi, C. Kruegel, and G. Vigna, Columbus: Android App Testing Through Systematic Callback Exploration, In Proceedings of the International Conference on Software Engineering (ICSE), 2023.

---

*Authors with equal contributions

4. **D. Das**, P. Bose, A. Machiry, S. Mariani, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, Hybrid Pruning: Towards Precise Pointer and Taint Analysis, In Proceedings of the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2022.
5. **D. Das**, P. Bose, N. Ruaro, C. Kruegel, and G. Vigna, Understanding Security Issues in the NFT Ecosystem, In Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2022.
6. P. Bose, **D. Das**, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, Sailfish: Vetting Smart Contract State-Inconsistency Bugs in Seconds, In Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P), 2022.
7. N. Redini, A. Continella, **D. Das**, G. D. Pasquale, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices, In Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P), 2021.
8. D. Song, F. Hetzelt, **D. Das**, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J. P. Seifert, and M. Franz, PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary, In Proceedings of the Network and Distributed System Security Symposium (NDSS), 2019.
9. N. Redini, A. Machiry, **D. Das**, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, BootStomp: On the Security of Bootloaders in Mobile Devices, In Proceedings of the USENIX Security Symposium (Usenix), 2017.
10. P. Bose, **D. Das**, and C. P. Rangan, Constant Size Ring Signature Without Random Oracle, In Proceedings of the Australasian Conference on Information Security and Privacy (ACISP), 2015.

Abstract

Finding Attacks and Vulnerabilities in Critical Systems

by

Dipanjan Das

Starting from that historic moment in 1948 when the first ever piece of software was written and successfully executed on a stored-program computer to this era of supercomputers, software have continuously been evolving in tandem with the underlying hardware to churn the last bit of performance out of the silicon. Long gone those days when the only use of software was to perform some simple calculations, much like today's handheld calculators. In the last few decades, the software industry has witnessed tremendous growth. The collective effort of the community has pushed software to its limit—both in terms of complexity and criticality. Today, software is frequently used in a multitude of critical applications, from solving existential problems to supporting diverse business scenarios. Traffic control systems, medical devices, nuclear power grids, the defense and military systems, autonomous vehicles, industrial control systems, the on-board computer of spacecrafts, financial trading systems—all these systems have one thing in common—even the most minor glitch in the software running on them can wreak havoc.

Given the variety of use-cases, deployment scenarios, framework or language used to develop the software, almost inevitably, no single technique is enough to deal with the complexity of analyzing critical software components. For example, a financial trading system runs in a very different environment than an operating system kernel, which would bring in different set of security concerns from a researcher's perspective. Similarly, the impact of failure of both the systems would be different as well. On the other hand,

an operating system kernel would be highly optimized for performance, which would, in turn, influence the choice of the language it would be written in. Despite these challenges, by the very nature of critical systems, the need of ensuring the safety and security of such systems is paramount.

My Ph.D. is inspired by the diversity, challenges, and the importance of such critical systems. In my research journey, I explored ways to understand, attack, and mitigate the threats on critical systems through the lens of a security researcher. In this thesis, I will first provide a detailed introduction of critical systems, along with the unique challenges in their security analysis, highlighting why a *one-size-fits-all* technique is likely not to work across systems. Then, I will present my research which pushes the limits of the current advancements in the security analysis for critical systems. Specifically, I will cover the following—**(i)** PERISCOPE, a technique to find vulnerabilities in the operating system kernel through a non-traditional attack surface. In the Wi-Fi drivers of two popular chipset vendors, PERISCOPE discovered 15 unique vulnerabilities, 9 of which were previously unknown. **(ii)** An in-depth analysis of the multi-billion dollar Non-Fungible Token (NFT) ecosystem, focusing on the security and privacy issues, and the design weaknesses found in the NFT marketplaces. In the top 8 marketplaces (ranked by transaction volume), we discovered a number of potential issues, many of which can lead to substantial financial losses, and finally **(iii)** Hybrid Pruning, a novel program analysis technique that injects run-time information in the traditional static analysis to improve its precision. On our dataset of 12 `CGC` and 8 real-world applications, our hybrid approach cuts down the warnings up to 21% over vanilla static analysis, while reporting 19 out of 20 bugs in total. For each approach, I will first present the technique, and then establish its real-world applicability through thorough evaluations.

Contents

Chapter 1

Introduction

Software has permeated every facet of our lives, from the moment we wake up until we go to bed. It powers our smartphones, manages our social interactions through social media platforms, and facilitates online shopping and banking transactions. It is the backbone of industries such as healthcare, finance, transportation, and entertainment. Yet, depending on the context the software is being used, some applications are more critical than the others. Critical software systems are those that are essential for the functioning and operation of the critical infrastructure, industries, or services. Critical software systems have a profound impact on our lives, influencing various aspects of modern society. For example, an air traffic control system enables safe air travel, while power grid management systems ensure a stable supply of electricity. Medical device software saves lives, and enhances patient care, while financial trading systems drive global economic activity. Without these critical software systems, our daily lives would be significantly disrupted, highlighting their immense importance in maintaining the functioning and progress of society. Since these systems are designed to perform crucial tasks, and have a direct impact on safety, security, or economic stability, they often require high levels of reliability, availability, and security. Due to their critical nature, these systems often undergo rigorous testing, certification, and security measures to mitigate risks, and ensure reliable operation.

1.1 Taxonomization

According to the literature of software dependability and reliability [1], there are four major types of critical systems:

- **Safety critical.** A system whose failure may lead to an injury, loss of life, or serious environmental damage. An example of a safety-critical system is a control system for a chemical manufacturing plant.
- **Mission critical.** A system whose failure may lead to the failure of some goal-directed activity, or disrupt the overall system or project objectives, *e.g.*, loss of critical infrastructure or data. An example of a mission-critical system is a navigational system for a spacecraft.
- **Business critical.** A system whose failure may lead to significant tangible, or intangible economic costs, *e.g.*, loss of business, or damage to reputation. Such a failure could result in very high costs for the business using that system. An example of a business-critical system is the customer accounting system in a bank, or a trading system run by a financial trading exchange, or a Non-Fungible Token (NFT) marketplace.
- **Security critical.** A system whose failure may lead to the loss of sensitive data through theft, or accidental loss. An example of a security-critical system is the operating system kernel.

It is worth mentioning that many systems have overlapping aspects of criticality, *e.g.*, a system might be both safety-critical and business-critical at the same time.

1.2 Attack surfaces and attacks

Critical systems are diverse, and expose large attack surfaces. So, the attacks on them are diverse as well. Attacks on them pose significant risks, and can have far-reaching consequences. Malicious actors, often economically motivated and funded by nefarious organizations, target these systems to disrupt essential services, compromise sensitive data, and exploit vulnerabilities for personal gain, or to cause harm. From sophisticated cyber-attacks on power grids and transportation systems to targeted attacks on healthcare and financial systems, the impact can be severe. These attacks may include techniques such as zero-day exploits, social engineering, ransomware, or distributed denial-of-service (DDoS) attacks. The potential outcomes can range from financial losses, and reputational damage to compromised safety, loss of life, and societal disruption. Since critical systems are diverse in nature, I will primarily focus on two types of systems in this dissertation—business-critical and security-critical systems.

Security-critical system. The OS kernel is a security-critical component that sits between the bare hardware and the user-space applications, and shields the applications from the complexity of low-level interactions by providing critical system services, *e.g.*, memory management, process management, user management, storage management, networking, interfacing with hardware devices, *etc.* Typically, it runs with an elevated privilege level enforced by the processor, *e.g.*, ring 0 on Intel architecture. Therefore, the compromise of a kernel component has a greater impact on the security of the entire system than any user-space component, for example, attacks on the kernel can lead to the compromise of sensitive data, unauthorized privilege escalation, system crashes, and the execution of arbitrary code in the extreme case. Worse, any compromise or exploitation of the kernel can have cascading effects, enabling attackers to gain control on any user-space application running atop.

OS kernels are found in computing devices of various sizes, processing abilities, applications, and complexities. General-purpose computing devices are often composed of an application processor, surrounded by an array of peripheral devices, *e.g.*, Wi-Fi, Bluetooth, and various sensors. While the peripherals run proprietary, special-purpose firmware; the application processor hosts the operating system (OS) kernel, usually Linux or its derivative, with the user applications sitting atop. With the recent proliferation of the peripherals present in such devices, both in types and numbers, the communication between these peripherals, and the OS kernel have become extremely complex and chaotic. A probing framework allows an analyst to probe, record, mutate or replay the data stream flowing both *to* and *from* the many peripherals. The OS kernel is an attractive target for remote attackers. If compromised, the kernel gives adversaries full system access, including the ability to install rootkits, extract sensitive information, and perform other malicious actions, all while evading detection. Most of the kernel's attack surface is situated along the system call boundary. Ongoing kernel protection efforts have focused primarily on securing this boundary; several capable analysis and fuzzing frameworks have been developed for this purpose.

However, there are additional paths to kernel compromise that do not involve system calls, as demonstrated by several recent exploits. For example, by compromising the firmware of a peripheral device such as a Wi-Fi chipset, and subsequently sending malicious inputs from the Wi-Fi chipset to the Wi-Fi driver, adversaries have been able to gain control over the kernel without invoking a single system call. Unfortunately, the lack of a practical probing and fuzzing frameworks that can help developers find and fix such vulnerabilities occurring along the hardware-OS boundary is concerning.

Business-critical system. An example of a business-critical system is a Non-Fungible Token (NFT) marketplace (NFTM). Non-Fungible Tokens (NFTs) have emerged as a way to collect digital art as well as an investment vehicle. Despite having been popularized only recently, NFT markets have witnessed several high-profile (and high-value) asset sales and a tremendous growth in trading volumes over the last year. Several NFT marketplaces (NFTMs), *e.g.*, OpenSea, Rarible, and Axie, emerged in recent years to facilitate buying and selling NFTs. This has sparked the interest of both crypto art collectors and traders. To put things into perspective, OpenSea, the largest NFTM, collected $236M USD in platform fees generated out of a trading volume of $3.5B USD [2] in August 2021 alone. This is around half of the volume [3] generated by the e-commerce giant eBay during the same period. And the all-time combined trading volume of the top three NFTMs—OpenSea, Axie, and CryptoPunks—surpassed $10B USD in September 2021 [4]. Individual NFT sales have also skyrocketed in recent months [5], with nine out of ten of the most expensive sales [6] taking place between February and August 2021. For example, the media widely reported on the digital artist Beeple, who sold an art piece for $69.3M USD; as another example, the first tweet of Twitter CEO Jack Dorsey was sold for $2.9M USD. Also, NFTMs have surfaced as the most *gas*-eating Ethereum contracts. For example, OpenSea made it to the top of the list of *gas-guzzlers* in Etherscan [7], consuming around 20% of the gas spent by the network.

As the NFT space exploded with multi-million dollar sales, cybercriminals and scammers have inevitably flocked to the markets to make quick profits and cheat unsuspecting users. As a result, numerous NFT scams also made recent headlines. Unfortunately, these marketplaces have not yet received much security scrutiny. Instead, most academic research has focused on attacks against decentralized finance (DeFi) protocols and automated techniques to detect smart contract vulnerabilities. With enormous funds flowing into *decentralized finance* (DeFi) applications, scams have become lucrative money-making opportunities. Previous research studied several different aspects of crypto-economic attacks, *e.g.*, financial repercussions due to transaction reordering [8–11], flash loan abuse [12], arbitrage opportunities [13], and pump-and-dump schemes [14–16]. Besides protocol attacks, there also exists a substantial body of work on automated detection of smart contract vulnerabilities, *e.g.*, reentrancy, transaction order dependence, integer overflows, and unhandled exceptions [17–29].

1.3 Automated vulnerability analysis

Analyzing attacks and finding vulnerabilities in critical systems pose certain challenges. The codebase of most of such systems is large, and ever-increasing in terms of lines of code. In case of the kernel, it also deals with the complex aspects of supporting a wide range of hardware and processor architectures, concurrency, and performance optimizations. Due to performance reasons, such code is often written in memory unsafe languages, like C and C++. As it is for any large legacy codebase, they can hardly benefit from the cutting-edge research on safe programming languages and their applications, because it would require a significant amount of developers' time and effort to port existing code to a new language and paradigm. The battle-tested tools in the researchers' arsenal, *i.e.*, most static (pointer and taint analysis) and dynamic (symbolic execution) analysis techniques succumb to either time or resource budget; thereby causing severe scalability issues.

The key techniques to automatically analyze large-scale software systems (which critical systems are a part of) are as follows:

Static analysis. Static program analysis analyzes a program without actually executing it, and look for potential errors in the program behavior. Pointer and taint analyses are the building blocks of several other static analysis techniques. Depending on the precision requirement, they can be flow-, context-, field-, or path-sensitive (in the extreme case). However, as the analysis aims for a higher degree of sensitivity or a composition of sensitivities, that makes them more memory and compute-intensive. Therefore, to make the analysis practical, these techniques frequently (and unfortunately) sacrifice *precision* in favor of *scalability* by over-approximating program behaviors by switching to a lower sensitivity, and making certain approximate choices. Scaling these analyses to real-world codebases written in memory-unsafe languages like the OS kernel, while retaining precision under the constraint of practical time and resource budgets is an open problem. Now, the imprecision from these basic analyses trickles down to their client analyses (*i.e.*, analyses that are dependent on them), thus making them imprecise as well, in turn. Symbolic execution is another static technique which collects constraints along a particular program path to derive the program input, or check for error conditions. It, too, suffers from the scalability issues due to path explosion.

Dynamic analysis. Dynamic program analysis is a technique to analyze a program's behavior during execution. Fuzzing is a popular dynamic technique where the program under test is typically fed with well-crafted, yet random data, and the program behavior is observed for the presence of bugs. A complex system like the OS kernel has an enormous state-space, which builds up over multiple inputs. Navigating through a state-space of this magnitude without understanding the semantics and context of a program is challenging for a fuzzer. System calls (syscalls) are the entry points to the OS kernel from the user-space. A OS fuzzer invokes those syscalls with random arguments. Now, those syscalls have mutual dependencies with each other. These dependencies along with the statefulness hinder the exploration of the kernel code by any automated technique.

Data analysis. Depending on the system in question, oftentimes it is useful for analyzing past attacks from a security research standpoint. Such analyses have two distinct benefits—**(i)** they provide actionable insights to develop signatures for future attacks, and **(ii)** they help developers close loopholes by making them aware of the past abuses. Fortunately, in the blockchain world, data (the blockchain itself) is publicly available. High-value financial attacks [30] are frequent in blockchain. But also, it brings in a unique opportunity for the researchers to study such attacks, propose and deploy mitigation, and develop analysis tools to equip the community better for similar future attacks. However, such analyses are not always straightforward; requiring deep understanding of the financial protocol, attack simulation, and attack signature identification; oftentimes across multiple protocols just to understand one single attack.

1.4 Contributions

In this dissertation, I present my research on two types of critical systems—Non-Fungible Token (NFT) marketplace, which is a business-critical system, and operating system (OS) kernel and application software, which are security-critical in their own right. The security analysis of critical systems is challenging for many reasons. First, the nature of interactions with critical systems vary across applications, which results in their attack surfaces and attackers' capabilities being different. For example, the attacker model involved in case of an OS kernel is very different from an economic attacker launching financial attacks on the NFT marketplaces. Second, such systems are developed using different languages and framework that have their own security concerns. For instance, OS kernels are highly performance-optimized, and therefore are typically written in a low-level language like `C`, which is memory-unsafe. On the other hand, in the blockchain world, they use languages like SOLIDITY, which has got a different set of issues like reentrancy. Third, critical systems are deployed in varied execution environments, *e.g.*, a OS kernel is highly multi-threaded, where as, the Ethereum virtual machine (EVM) follows a linear execution model. Given the disparities, no single technique is evidently not enough for the security analysis of the critical systems. What we rather need are specialized techniques tailored to a particular class of critical system. In my Ph.D., I demonstrated how traditional static and dynamic analysis can be used to automatically find vulnerabilities in critical systems, and how we can derive meaningful insights on the past attacks on such systems by a careful data analysis. In particular, this dissertation makes the following contributions:

- We develop PeriScope, a generic probing framework that can inspect the interactions between a driver and its corresponding device. PeriScope provides the means to analyze the hardware-OS boundary, and to build more specialized analysis tools. We extend PeriScope to build PeriFuzz, a vulnerability discovery tool tailored to detect driver vulnerabilities occurring along the hardware-OS boundary. The tool demonstrates the power of the PeriScope framework, and it systematizes the exploration of the hardware-OS boundary. PeriFuzz fuzzes overlapping fetches in addition to non-overlapping fetches, and warns about overlapping fetches that occurred before a driver crash. A warning observed before a driver crash may indicate the presence of double-fetch bugs. As part of our evaluation, we discovered previously known and unknown vulnerabilities in the Wi-Fi drivers of two of the most prominent vendors in the market. We responsibly disclosed relevant details to the corresponding vendors. We open-sourced our tool[*], in order to facilitate further research exploration of the hardware-OS boundary.
- To the best of our knowledge, we are the first to study the market dynamics and security issues of the multi-billion dollar NFT ecosystem. We systematize the NFT ecosystem, looking at the participating actors—NFT marketplaces (NFTM),

---

[*] https://github.com/securesystemslab/periscope

external entities, and users—and we analyze their mutual interactions. We leverage multiple sources of data, including the Ethereum blockchain, as well as asset and event data sourced from the NFTM dApps, to paint a holistic picture of how the ecosystem operates. We identify flaws in the NFTM designs, which, if abused, pose a significant financial risk. We identify the off-chain external entities connected to the NFT ecosystem, and how such entities can pose threats to users. We discover and quantify trading malpractices, such as wash trading, shill bidding, and bid shielding, which are taking place in the top marketplaces. The insight drawn from our analysis sheds light on some of the prime factors responsible for driving up the recent NFT frenzy. Interestingly, our findings show that at least half of such sales show some suspicious signs. We open-sourced our analysis framework[†] along with the data we collected to help researchers uncover further interesting insights about the emerging NFT economy.

- We propose *hybrid pruning*, a new hybrid program analysis technique that combines dynamic information with the vanilla static analysis to develop precise pointer and taint analyses. To demonstrate the effectiveness of our hybrid technique, we have further developed a vulnerability detection system as a client of our improved pointer and taint analyses. It exhibits significantly lower false positive rate as compared to its static counterpart. We implement our approach in a practical prototype, and show its efficacy in an experimental evaluation on two different datasets, *i.e.*, `CGC` [31], and a collection of popular real-world programs.

1.5 Thesis organization

The rest of this thesis is structured as follows.

[†] https://github.com/ucsb-seclab/nft-security-study

In Chapter 2, I present PeriScope, a Linux kernel based probing framework that enables fine-grained analysis of device-driver interactions. PeriScope hooks into the kernel's page fault handling mechanism to either passively monitor and log traffic between device drivers and their corresponding hardware, or mutate the data stream on-the-fly using a fuzzing component, PeriFuzz, thus mimicking an active adversarial attack. PeriFuzz accurately models the capabilities of an attacker on peripheral devices, to expose different classes of bugs including, but not limited to, memory corruption bugs and double-fetch bugs. To demonstrate the risk that peripheral devices pose, as well as the value of our framework, we have evaluated PeriFuzz on the Wi-Fi drivers of two popular chipset vendors, where we discovered 15 unique vulnerabilities, 9 of which were previously unknown.

In Chapter 3, we first present a systematic overview of how the NFT ecosystem works, and we identify three major actors: marketplaces, external entities, and users. We then perform an in-depth analysis of the top 8 marketplaces (ranked by transaction volume) to discover potential issues, many of which can lead to substantial financial losses. We also collected a large amount of asset and event data pertaining to the NFTs being traded in the examined marketplaces. We automatically analyze this data to understand how the entities external to the blockchain are able to interfere with NFT markets, leading to serious consequences, and quantify the malicious trading behaviors carried out by users under the cloak of anonymity. Finally, we studied the 15 most expensive NFT sales to date, and discovered discrepancies in at least half of these transactions.

In Chapter 4, we present a novel technique called *hybrid pruning*, where we inject the information collected from a program's dynamic trace, which is accurate by its very nature, into a static pointer or taint analysis system to enhance its precision. We also tackle the challenge of combining static and dynamic analyses, which operate in two different analysis domains, in order to make the interleaving possible. Finally, we show the usefulness of our approach by reducing the false positives emitted by a static vulnerability detector that consumes the improved points-to and taint information. On our dataset of 12 `CGC` and 8 real-world applications, our hybrid approach cuts down the warnings up to 21% over vanilla static analysis, while reporting 19 out of 20 bugs in total.

Finally, I conclude my thesis by presenting the collective insight drawn from my research in Chapter 5.

Chapter 2

PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary

Modern electronics often include subsystems manufactured by a variety of different vendors. For example, in a modern cellphone, besides the main application processor running a smartphone operating system such as Android, one might find a number of *peripheral devices* such as a touchscreen display, camera modules, and chipsets supporting various networking protocols (cellular, Wi-Fi, Bluetooth, NFC, etc.). Peripheral devices by different manufacturers have different inner workings, which are often proprietary. *Device drivers* bridge the gap between stable and well-documented operating system interfaces on one side and peripheral devices on the other, and make the devices available to the rest of the system.

Device drivers are privileged kernel components that execute along two different trust boundaries of the system. One of these boundaries is the system call interface, which exposes kernel-space drivers to user-space adversaries. The *hardware-OS interface* should also be considered a trust boundary, however, since it exposes drivers to potentially compromised peripheral hardware. These peripherals should not be trusted, because they may provide a remote attack vector (e.g., network devices may receive malicious packets over the air), and they typically lack basic defense mechanisms. Consequently, peripheral devices have frequently fallen victim to *remote* exploitation [32–37]. Thus, a device driver must robustly enforce the hardware-OS boundary, but programming errors do occur. Several recently published attacks demonstrated that peripheral compromise can be turned into full system compromise (i.e., remote kernel code execution) by coaxing a compromised device into generating specific outputs, which in turn trigger a vulnerability when processed as an input in a device driver [38, 39].

The trust boundary that separates peripheral subsystems from kernel drivers is therefore of great interest to security researchers. We present PeriScope, which to our knowledge is the first generic framework that facilitates the exploration of this boundary. PeriScope focuses on two popular device-driver interaction mechanisms: *memory-mapped I/O* (MMIO) and *direct memory access* (DMA). The key idea is to monitor MMIO or DMA mappings set up by the driver, and then dynamically *trap* the driver's accesses to such memory regions. PeriScope allows developers to register hooks that it calls upon each trapped access, thereby enabling them to conduct a fine-grained analysis of device-driver interactions. For example, one can implement hooks that record and/or mutate device-driver interactions in support of reverse engineering, record-and-replay, fuzzing, etc.

To demonstrate the risk that peripheral devices pose, as well as to showcase versatility of the PeriScope framework, we created PeriFuzz, a driver fuzzer that simulates attacks originating in untrusted, compromised peripherals. PeriFuzz traps the driver's *read accesses* to MMIO and DMA mappings, and fuzzes the values being read by the driver. With a compromised device, these values should be considered to be under an attacker's control; the attacker can freely modify these values at any time, even in between the driver's reads. If the driver reads the same memory location multiple times (*i.e.*, overlapping fetches [40]) while the data can still be modified by the device, double-fetch bugs may be present [41, 42]. PeriFuzz accurately models this adversarial capability by fuzzing not only the values being read from different memory locations, but also ones being read from the same location multiple times. PeriFuzz also tracks and logs all overlapping fetches and warns about ones that occurred before a driver crash to help identify potential double-fetch bugs.

Existing work on analyzing device-driver interactions typically runs the entire system including device drivers in a controlled environment [43–50], such as QEMU [51] or S2E [52]. Enabling analysis in such an environment often requires developer efforts tailored to specific drivers or devices, e.g., implementing a virtual device or annotating driver code to keep symbolic execution tractable. In contrast, PeriScope uses a page fault based *in-kernel monitoring* mechanism, which works with all devices and drivers in their existing testing environment. As long as the kernel gets recompiled with our framework, PeriScope and PeriFuzz can analyze device-driver interactions with relative ease, regardless of whether the underlying device is virtual or physical, and regardless of the type of the device. Extending our framework is also straightforward; for example, PeriFuzz accepts any user-space fuzzer, *e.g.*, AFL, as a plug-in, which significantly reduces the engineering effort required to implement proven fuzzing strategies [53–57].

We validated our system by running experiments on the software stacks shipping with the Google Pixel 2 and the Samsung Galaxy S6, two popular smartphones on the market at the time of development. To simulate remote attacks that would occur *over the air* in a real-world scenario, we focused on the Wi-Fi drivers of these phones in evaluating our framework. The Google Pixel 2 and Samsung Galaxy S6 are equipped with Qualcomm and Broadcom chipsets, respectively. These two are arguably the most popular Wi-Fi chipset manufacturers at the time of our experiments. In our experiments, our system identified 15 unique vulnerabilities in two device drivers, out of which 9 vulnerabilities were previously unknown, and 8 new CVEs were assigned. We have reported the discovered vulnerabilities to the respective vendors and are working with them on fixing these vulnerabilities. We hope that our tool will aid developers in hardening the hardware-OS boundary, leading to better software security.

2.1 Background

In this section, we provide the technical background necessary to understand how peripheral devices interact with the OS. We also discuss isolation mechanisms that allow the OS to protect itself against misbehaving peripherals, as well as tools to analyze hardware-OS interactions.

2.1.1 Hardware-OS Interaction

Figure 2.1 illustrates the various ways in which devices can interact with the OS and the device driver. Although we assume that the device driver runs on a Linux system with an ARMv8-A/AArch64 CPU, the following discussion generally applies to other platforms as well.

Interrupts

A device can send a signal to the CPU by raising an interrupt request on one of the CPU's interrupt lines. Upon receiving an interrupt request, ARMv8-A CPUs first mask the interrupt line so that another interrupt request cannot be raised on the same line while the first request is being handled. Then, the CPU transfers control to the interrupt

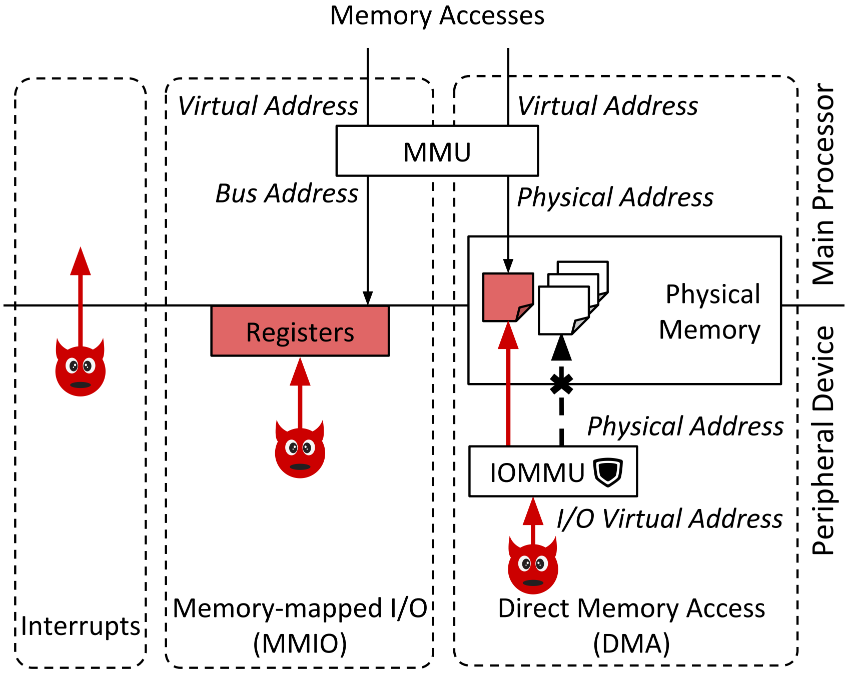


Figure 2.1: Hardware-OS interaction mechanisms

handler registered by the OS for that interrupt line. Interrupt handlers can be configured at any time, though the OS typically configures them at boot time.

Processing Interrupts To maximize the responsiveness and concurrency of the system, the OS attempts to defer interrupt processing so that the interrupt handler can return control to the CPU as soon as possible. Typically, interrupt handlers only process interrupts in full if they were caused by time-sensitive events or by events that require immediate attention. All other events are processed at a later time, outside of the interrupt context. This mechanism is referred to as top-half and bottom-half interrupt processing in Linux lingo.

In Linux, after performing minimal amount of work in the hardware interrupt context (`hardirq`), the device driver schedules the work to be run in either software interrupt context (`softirq`), kernel worker threads, or the device driver's own kernel threads, based on its priority. For higher priority work, a device driver can register its own `tasklet`, a deferred action to be executed under the software interrupt context, which also ensures serialized execution. Lower priority work can further be deferred either to kernel worker threads (using the workqueue API) or to the device driver's own kernel threads.

Memory-Mapped I/O

Analogous to peripherals using interrupts to signal the OS and the device driver, the CPU uses memory-mapped I/O (MMIO) to signal peripherals. MMIO maps a range of kernel-space virtual addresses to the hardware registers of peripheral devices. This allows the CPU to use normal memory access instructions (as opposed to special I/O instructions) to communicate with the peripheral device. The CPU observes such memory accesses and redirects them to the corresponding hardware. In Linux, device drivers call `ioremap` to establish an MMIO mapping, and `iounmap` to remove it.

Direct Memory Access

Direct memory access (DMA) allows peripheral devices to access physical memory directly. Typically, the device transfers data using DMA, and then signals the CPU using an interrupt. There are two kinds of DMA buffers: coherent and streaming.

Coherent DMA buffers (also known as consistent DMA buffers) are usually allocated and mapped only once at the time of driver initialization. Writes to coherent DMA buffers are usually uncached, so that values written by either the peripheral processor or the CPU are immediately visible to the other side.

Streaming DMA buffers are backed by the CPU's cache, and have an explicit owner. They can either be owned by the CPU itself, or by one of the peripheral processors. Certain kernel-space memory buffers can be "mapped" as streaming DMA buffers. However, once a streaming DMA buffer is mapped, the peripheral devices automatically acquires ownership over it, and the kernel can no longer write to the buffer. Unmapping a streaming DMA buffer revokes its ownership from the peripheral device, and allows the CPU to access the buffer's contents. Streaming DMA buffers are typically short-lived, and are often used for a single data transfer operation.

2.1.2 Input/Output Memory Management Unit

Since DMA allows peripherals to access physical memory directly, its use can be detrimental to the overall stability of the system if a peripheral device misbehaves. Modern systems therefore deploy an input output memory management unit (IOMMU) (also known as system memory management unit, or SMMU, on the ARMv8-A/AArch64 architecture) to limit which regions of the physical memory each device can access. Similar to the CPU's memory management unit (MMU), the IOMMU translates device-visible virtual addresses (*i.e.*, I/O addresses) to physical addresses. The IOMMU uses translation tables, which are configured by the OS prior to initiating a DMA transfer. Device-initiated accesses that fall outside of the translation table range will trigger faults that are visible to the OS.

2.1.3 Analyzing Hardware-OS Interaction

Vulnerabilities in device drivers can lead to a compromise of the entire system, since many of these drivers run in kernel space. To detect these vulnerabilities, driver developers can resort to dynamic analysis tools that monitor the driver's behavior and report potentially harmful actions. Doing this ideally requires insight into the communication between the driver and the device, as this communication can provide the context necessary to find the underlying cause of a vulnerability. Analyzing device-driver communication requires (i) an instance of the device, whether physical or virtual, and (ii) a monitoring mechanism to observe and/or influence device-driver communication. Existing approaches can therefore be classified based on where and how they observe (and possibly influence) device-driver interactions.

Device Adaptation To exercise direct control over the data sent from the hardware to the driver, an analyst can adapt the firmware of real devices to include such capabilities. This can be done by reverse engineering the firmware and reflashing a modified one [58], or by using custom hardware that supports reprogramming of devices [59]. However, these frameworks are typically tailored to specific devices, and given the heterogeneity of peripheral devices, their applicability is limited. For example, Nexmon only works for some Broadcom Wi-Fi devices [58], and Facedancer11, a custom Universal Serial Bus (USB) device, can only analyze USB device drivers [59].

Virtual Machine Monitor A driver can be tested in conjunction with virtual devices running in a virtual environment such as QEMU [51]. The virtual machine monitor observes the behavior of its guest machines and can easily support instrumentation of the hardware-OS interface. Previous work uses existing implementations of virtual devices for testing the corresponding drivers [45,46]. For many devices, however, an implementation of a virtual device does not exist. In this case, developers must manually implement a virtual version of their devices to interact with the device driver they wish to analyze [43]. Several frameworks alleviate the need for virtual devices by relaying I/O to real devices [60, 61], but these frameworks generally require a non-trivial porting effort for each driver and device, and/or do not support DMA.

Symbolic Execution S2E augments QEMU with selective symbolic execution [52]. Several tools leverage S2E to analyze the interactions between OS kernel and hardware by selectively converting hardware-provided values into symbolic values [47–50]. However, symbolic execution in general is prohibitively slow due to the path explosion and constraint solving problem. Moreover, symbolic execution itself does not reveal vulnerabilities, but rather generates a set of constraints that must be analyzed by separate checkers. Writing such a checker is not trivial. Most of the checkers supported by SymDrive, for example, target stateless bugs such as kernel API misuses, but ignore memory corruption bugs [49].

2.2 PeriScope Design

We designed PeriScope as a dynamic analysis framework that can be used to examine bi-directional communication between devices and their drivers over MMIO and DMA. Contrary to earlier work on analyzing device-driver communication on the device side, we analyze this communication on the driver side, by intercepting the driver's accesses to communication channels. PeriScope does this by hooking into the kernel's page fault handling mechanism. This design choice makes our framework driver-agnostic; PeriScope can analyze drivers with relative ease, regardless of whether the underlying device is virtual or real, and regardless of the type of the peripheral device.

At a high level, PeriScope works as follows. First, PeriScope automatically detects when the target device driver creates a MMIO or DMA memory mapping, and registers it. Then, the analyst selects the registered mappings that he/she wishes to monitor. PeriScope marks the pages backing these monitored mappings as not present in the kernel page tables. Any CPU access to those marked pages therefore triggers a page fault, even though the data on these pages is present in physical memory.

When a kernel page fault occurs, PeriScope first marks the faulting page as present in the page table (❶ in Figure 2.2). Then, it determines if the faulting address is part of any of the monitored regions (❷). If it is not, PeriScope re-executes the faulting instruction (➎), which will now execute without problems. Afterwards, PeriScope marks the page as not present again (➐), and resumes the normal execution of the faulting code.

If the faulting address does belong to a monitored region, PeriScope invokes a pre-instruction hook function registered by the user of the framework, passing information about the faulting instruction (➍). Then, PeriScope re-executes the faulting instruction (➎). Finally, PeriScope invokes the post-instruction hook registered by the driver (➏), marks the faulting page as not present again (➐), and resumes the execution of the faulting code.

2.2.1 Memory Access Monitoring

Tracking Allocations PeriScope hooks the kernel APIs used to allocate and deallocate DMA and MMIO regions*. We use these hooks to maintain a list of all DMA/MMIO allocation contexts and their active mappings. PeriScope assigns an identifier to every context in which a mapping is allocated, and presents the list of all allocation contexts as well as their active mappings to privileged user-space programs through the `debugfs` file system.

Enabling Monitoring PeriScope exposes a privileged user-space API that enables monitoring of DMA/MMIO regions on a per-allocation-context basis. Once monitoring is enabled for a specific allocation context, PeriScope will ensure that accesses to all current and future regions allocated in that context trigger page faults.

Clearing Page Presence PeriScope marks all pages containing monitored regions as not present in the kernel's page tables to force accesses to such pages to trigger page

---

*Establishing DMA and MMIO mappings is a highly platform-dependent process, so device drivers are obliged to use the official kernel APIs to do so.

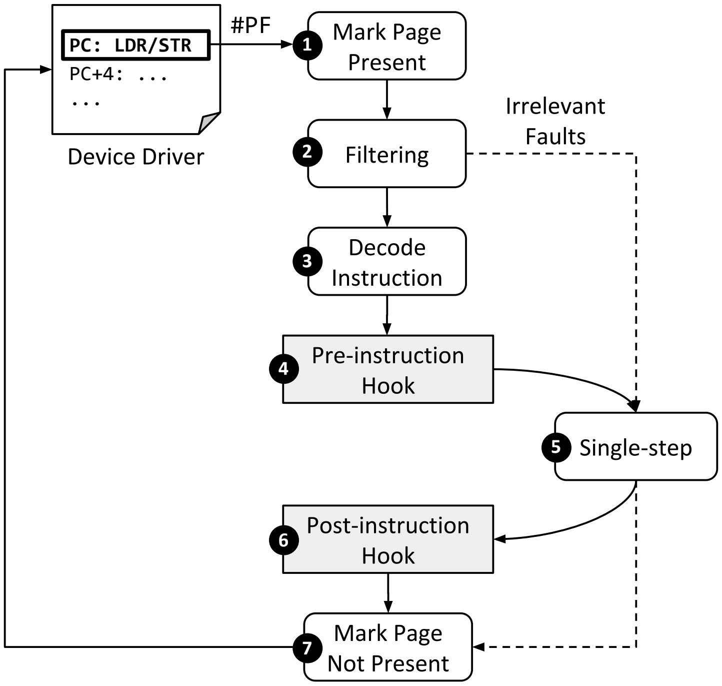


Figure 2.2: PeriScope fault handling

faults. One complication that can arise here is that modern architectures, including x86-64 and AArch64, can support multiple page sizes within the same page table. On AArch64 platforms, a single page table entry can serve physical memory regions of 4KB, 16KB, or 64KB, for example. If a single (large) page table entry serves both a monitored and a non-monitored region, then we split that entry prior to marking the region as not present. We do this to avoid unnecessary page faults for non-monitored regions. Note that, even after splitting page table entries, PeriScope cannot rule out spurious page faults completely, as some devices support DMA/MMIO regions that are smaller than the smallest page size supported by the CPU.

Trapping Page Faults PeriScope hooks the kernel's default kernel page fault handler to monitor page faults. Inside the hook function, we first check if the fault originated from a page that contains one of the monitored regions. If the fault originated from some other page, we immediately return from the hook function with an error code and defer the fault handling to the default page fault handler. If the fault did originate from a page containing a registered buffer, PeriScope marks that page as present (❶), and then checks if the faulting address falls within a monitored region (❷). If the faulting address is outside a monitored region, we simply single-step the faulting instruction (❺), mark the faulting page as not present again (❼), and resume normal execution of the faulting code. If the faulting address does fall within a monitored region, however, we proceed to the instruction decoding step (❸).

Instruction Decoding In order to accurately monitor and (potentially) manipulate the communication between the hardware/firmware and the device driver, we need to extract the source register, the destination register and the access width of the faulting instruction (❸ in Figure 2.2). We implemented a simple AArch64 instruction decoder, which provides this information for all load and store instructions. PeriScope carries this information along the rest of its fault handling pipeline.

Pre-instruction Hook After decoding the instruction, PeriScope calls the pre-instruction hook that the user of our framework can register (❹). We pass the address of the faulting instruction, the memory region type (MMIO or DMA coherent/streaming), the instruction type (load or store), the destination/source register, and the access register width to this hook function. The pre-instruction hook function can return two values: a default value and a skip-single-step value. If the function returns the latter, PeriScope proceeds immediately to step ❻. Otherwise, PeriScope proceeds to step ❺.

PeriScope provides a default pre-instruction hook which logs all memory stores before the value in the source register is stored to memory. We maintain this log in a kernel-space circular buffer that can later be read from the file system using `tracefs`.

Single-stepping When execution returns from the pre-instruction hook, and the hook function did not return the skip-single-step value, we re-execute the faulting instruction, which can now access the page without faulting. We use the processor's single-stepping support to ensure that only the faulting instruction executes, but none of its successors do (❺).

Post-instruction Hook When PeriScope regains control after single-stepping, it first clears the page present flag for the faulting page again so that future accesses to the faulting page once again trigger a page fault. Then, it calls the post-instruction handler, which, similarly to the pre-instruction handler, has a default implementation that can be overridden through our API (❻). The default handler logs all memory loads by examining and logging the value that is now stored in the destination register.

2.3 PeriFuzz Design

We built PeriFuzz as a client module for PeriScope. PeriFuzz can generate and provide inputs for device drivers. The goal of our fuzzer is to uncover vulnerabilities that could potentially be exploited by a compromised peripheral device.

2.3.1 Threat Model

Peripheral Compromise We assume that the attacker can compromise a peripheral, which, in turn, can send arbitrary data to its device driver. Compromising a peripheral device is feasible because such devices rarely deploy hardware protection mechanisms or software mitigations. As a result, silent memory corruptions occur frequently [62], which significantly lowers the bar to mount an attack. That peripherals can turn malicious after being attacked was demonstrated by successful *remote* compromises of several network devices such as ethernet adapters [32], GSM baseband processors [33, 37], and Wi-Fi processors [34–36].

IOMMU/SMMU Protection For many years, a strict hardware-OS security boundary existed in theory, but it was not enforced in practice. Most device drivers *trusted* that the peripheral was benign, and gave the device access to the entire physical memory (provided that the device was DMA-capable), thus opening the door to DMA-based attacks and rootkits [63, 64]. This situation has changed for the better with the now widespread deployment of IOMMU units (or SMMU for AArch64). IOMMUs can prevent the device from accessing physical memory regions that were not explicitly mapped by the MMU, and they prevent peripherals from accessing streaming DMA buffers while these are mapped for CPU access. The latter restriction can be imposed by invalidating IOMMU mappings, or by copying the contents of a streaming DMA buffer to a temporary buffer (which the peripheral cannot access) before the CPU uses them [65, 66]. We assume that such an IOMMU is in place, and that is being used correctly.

Summary In our model, the *attacker* can (i) compromise a peripheral such as a Wi-Fi chipset over the air by abusing an existing bug in the peripheral's firmware, (ii) *exercise control* over the compromised peripheral to send arbitrary data to the device driver, and, (iii) not access the main physical memory, except for memory regions used for communicating with the device driver.

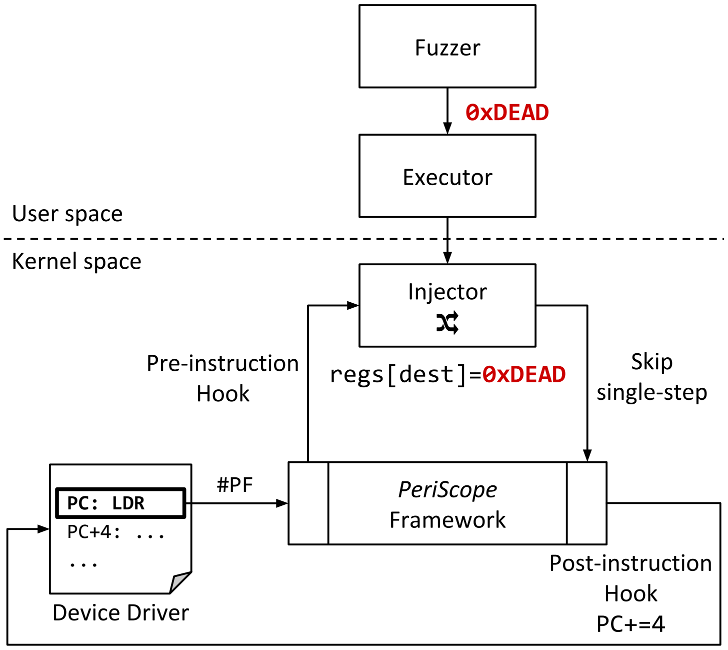


Figure 2.3: PeriFuzz overview

2.3.2 Design Overview

PeriFuzz is composed of a number of components, as illustrated in Figure 2.3. Our design is fully modular, so each component can be swapped out for an alternative implementation that exposes the same interface.

Fuzzer We use a fuzzer that runs in user space. This component is responsible for generating inputs for the device driver and processing execution feedback. Our modular design allows us to use any fuzzer capable of fuzzing user-space programs. We currently use AFL as our fuzzer, as was done in several previous works that focus on fuzzing kernel subsystems [67–69].

Executor The executor is a user-space-resident bridge between the fuzzer (or any input provider) and the injector. The executor takes an input file as an argument, and sends the file content to the injector via a shared memory region mapped into both the executor's and the injector's address spaces. The executor then notifies the injector that the input is ready for injection, and periodically checks if the provided input has been consumed. PeriFuzz launches an instance of the executor for every input the fuzzer generates. The executor is also used to reproduce a crash by providing the last input observed before the crash.

Injector The injector is a kernel-space module that interfaces with our PeriScope framework. The injector registers a pre-instruction hook with PeriScope, which allows the injector to monitor and manipulate all data the device driver receives from the device. At every page fault, the injector first checks if fuzzing is currently enabled, and if there is a fuzzer/executor-provided input that has not been consumed yet. If both conditions are met, the injector overwrites the destination register with the input generated by the fuzzer.

Note that PeriFuzz manipulates only the values device drivers *read* from MMIO and DMA mappings, but not the values they write. PeriFuzz, in other words, models compromised devices, but not compromised drivers.

2.3.3 Fuzzer Input Consumption

We treat each fuzzer-generated input as a serialized sequence of memory accesses. In other words, our injector always consumes and injects the first non-consumed inputs found in the input buffer shared between the executor and injector. This fuzzer input consumption model allows for *overlapping fetch fuzzing* as it automatically provides different values for multiple accesses to the same offsets within a target mapping (*i.e.*, overlapping fetches [40]). Providing different values for overlapping fetches enables us to find double-fetch bugs, if triggering such bugs leads to visible side-effects such as a driver crash. Our fuzzer also keeps track of the values returned for overlapping fetches, and can

Algorithm 1 Fuzzer Input Consumption at Each Driver Read

```
global variables                                  ▷ Initialized when switching fuzzer input
    $Input \leftarrow [...]$
    $InputOffset \leftarrow 0$
    $PrevReads \leftarrow \{\}$
    $OverlappingFetches \leftarrow \{\}$
end global variables
function FUZZDRIVERREAD($Address, Width, Type$)
    $Value \leftarrow Input[range(InputOffset, Width)]$
    for all $Prev$ in $PrevReads$ do
        $Overlap \leftarrow Prev.range \cap range(Address, Width)$
        if $Overlap$ is not empty then
            if $Type$ is DMA Streaming then
                $Value[Overlap] \leftarrow Prev.value(Overlap)$
            else
                $OverlappingFetches \leftarrow OverlappingFetches \cup \{(Overlap, Value)\}$
            end if
        end if
    end for
    $InputOffset \leftarrow InputOffset + Width$
    $PrevReads \leftarrow PrevReads \cup \{(Address, Width, Value)\}$
    return $Value$
end function
```

output this information when a driver crashes, thereby helping us to narrow down the cause of the crash. In fact, the double-fetch bugs we identified using PeriFuzz would not have been found without this information (see Section 2.5).

Since we assume that the attacker cannot access streaming DMA buffers while they are mapped for CPU access (see Section 2.3.1), we take extra care not to enable overlapping fetch fuzzing for streaming DMA buffers. To this end, we maintain a history of read accesses, and consult this history to determine if a new access overlaps with any previous access. If they overlap, we return the same values returned for the previous access, and do not consume any bytes from the fuzzer input. Algorithm 1 shows how we pick values to inject for each driver read from an MMIO or DMA mapping.

An additional benefit of our fuzzer input consumption model is that it helps to keep the input size small, because we only have to generate fuzzer input bytes for read accesses that actually happen and not for the entire fuzzed buffer, which may contain bytes that are never read.

2.3.4 Register Value Injection

PeriScope provides the destination register and the access width when it calls into PeriFuzz's pre-instruction hook handler. The fuzzer input is consumed for that exact access width, and then injected into the destination register. Our pre-instruction hook function returns the skip-single-step value to PeriScope (see Section 2.2.1), as we have emulated the faulting load instruction by writing a fuzzed value into its destination register. Our post-instruction hook function increments the program counter, so the execution of the driver resumes from the instruction that follows the fuzzed instruction.

2.3.5 Fuzzing Loop

Each iteration of the fuzzing loop consumes a single fuzzer-generated input. We align each iteration of the fuzzing loop to the software interrupt handler, *i.e.*, `do_softirq`. We do not insert hooks into the hardware interrupt handler, since work is barely done in the hardware interrupt context. The two hooks inserted before and after the software interrupt handler demarcate a single iteration of the fuzzing loop, in which PeriFuzz consecutively consumes bytes in a single fuzzer input. This design decision allows us to remain device-agnostic, but device driver developers could provide an alternative device-specific definition of an iteration by inserting those two hooks in their drivers. Several low priority tasks are often deferred to the device driver's own kernel threads, and the fuzzing loop can be aligned to the task processing loop inside those threads.

2.3.6 Interfacing with AFL

We use AFL [70], a well-known coverage-guided fuzzer, as PeriFuzz's fuzzing front-end. This is in line with previous work on fuzzing various kernel subsystems [67–69]. To fully leverage AFL's coverage-guidance, we added kernel coverage and seed generation support in PeriFuzz.

Coverage-guidance We modified and used KCOV to provide coverage feedback while executing inputs [71]. Existing implementations of KCOV were developed for fuzzing system calls and only collect coverage information for code paths reachable from system calls. To enable device driver fuzzing, we extended KCOV with support for collecting coverage information for code paths reachable from interrupt handlers. We also applied a patch to force KCOV to collect edge coverage information rather than basic block coverage information [72]. To collect coverage along the execution of the device driver, it is first compiled with coverage instrumentation. This instrumentation informs KCOV of hit basic blocks, which KCOV records in terms of edge coverage. The executor component retrieves the coverage feedback from kernel, once the input has been consumed. Then the executor copies this coverage information to a memory region shared with the parent AFL fuzzer process, after which we signal KCOV to clear the coverage buffer for the next fuzzing iteration.

Automated Seed Generation Starting with valid test cases rather than fully random inputs improves the fuzzing efficiency, as this lowers the number of input mutations required to discover new paths. To collect an initial seed of valid test cases, we use our PeriScope framework to log all accesses to a user-selected set of buffers. We provide an access log parser that automatically turns a sequence of accesses into a seed file according to our fuzzing input consumption model (see Section 2.3.3). That said, this

Table 2.1: LoC modified in the Linux kernel code and the PeriScope framework itself

Description	LoC
Linux DMA and MMIO allocation/deallocation APIs	92
Linux kernel page fault and debug exception handlers	46
PeriScope framework	3843

step is optional; one could start from any arbitrary seed, or craft test cases on their own.

2.4 Implementation

2.4.1 PeriScope

We based our implementation of PeriScope on Linux kernel 4.4 for AArch64. Our framework is, for the most part, a standalone component that can be ported to other versions of the Linux kernel and even to vendor-modified custom kernels with relative ease. The kernel changes required for PeriScope are relatively small compared to the framework implementation itself as shown in Table 2.1.

Tracking Allocations PeriScope hooks the generic kernel APIs used to allocate/deallocate MMIO and DMA regions to maintain a list of allocation contexts. We insert these hooks into the `dma_alloc_coherent` and `dma_free_coherent` functions to track coherent DMA mappings, into the `dma_unmap_page` function[†] and `dma_map_page` to track streaming DMA mappings, and into `ioremap` and `iounmap` to track MMIO mappings.

PeriScope assigns a context identifier to every MMIO and DMA allocation context. This context identifier is the XOR-sum of all call site addresses that are on the call stack at allocation time. We mask out the upper bits of all call site addresses to ensure that

---

[†] `dma_unmap_page` unmaps a streaming DMA mapping from the peripheral processor. Doing so transfers ownership of the mapping to the device driver.

Table 2.2: PeriFuzz implementation LoC

Component		LoC
Injector	Kernel-space	441
KCOV (modification)	Kernel-space	176
Executor	User-space	338
Python manager and utility scripts	Host	924

context identifiers remain the same across reboots on devices that enable kernel address space layout randomization (KASLR).

Monitoring Interface PeriScope provides a user-space interface by exposing `debugfs` and `tracefs` file system entries. Through this interface, a user can list all allocation contexts and their active mappings, enable or disable monitoring, and read the circular buffer where PeriScope logs all accesses to the monitored mappings.

As streaming DMA buffer allocations can happen in interrupt contexts, we use a non-blocking spinlock to protect access to data structures such as the list of monitored mappings. When accessing these data structures from an interruptible code path, we additionally disable interrupts to prevent interrupt handlers from deadlocking while trying to access the same structures.

2.4.2 PeriFuzz

We built PeriFuzz as a client for PeriScope. Table 2.2 summarizes the code we added or changed for PeriFuzz.

Kernel-User Interface The injector registers a device node that exposes device-specific `mmap` and `ioctl` system calls to the user-space executor. The executor can therefore create a shared memory mapping via `mmap` to the `debugfs` file exported by

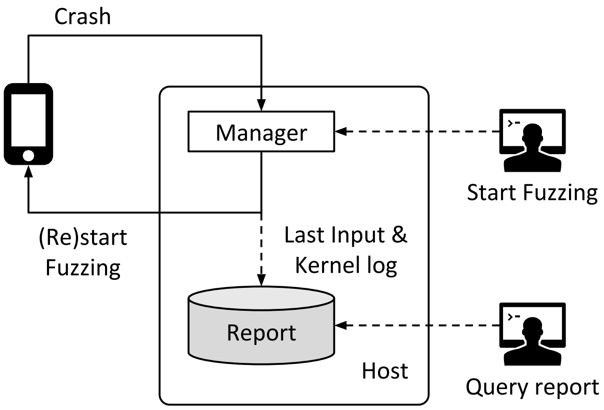


Figure 2.4: Continuous fuzzing with PeriFuzz

the injector module. Through this interface, the executor passes the fuzzer input to the injector running in the kernel space. The `ioctl` handler of the injector module allows the executor (i) to enable and disable fuzzing, and (ii) to poll the consumption status of a fuzzer input it provided. Similarly, KCOV provides the coverage feedback by exporting another `debugfs` file such that the executor can read the feedback by `mmap`ing the exported `debugfs` file.

Persisting Fuzzer Files Many fuzzers including AFL store meta-information about fuzzing and input corpus in the file system. However, these files might not persist if the kernel crashes before the data is committed to the disk. To avoid this, we ensure that all the fuzzer files are made persistent, by modifying AFL to call `fsync` after all file writes. Persisting all files allows us (i) to investigate crashes using the last crashing input and (ii) to resume fuzzing with the existing corpus stored in the file system.

Fuzzing Manager The fuzzing procedure is completely automated through Python scripts that run on a host separate from the target device. The continuous fuzzing loop is driven by a Python program, as illustrated in Figure 2.4. The manager process runs in a loop in which it (i) polls the status of the fuzzing process, (ii) starts/restarts fuzzing

Table 2.3: Target smartphones

	Google Pixel 2	Samsung Galaxy S6
Model Name	walleye	SM-G920F
Released	October, 2017	April, 2015
SoC	Snapdragon 835	Exynos 7420
Kernel Version	4.4	3.10
Wi-Fi Device Driver	`qcacld-3.0`	`bcmdhd4358`
Wi-Fi IOMMU Protection	Yes	No

if required, (iii) detects device reboots, (iv) downloads the kernel log and the last input generated before the crash after a reboot, and (v) examines the last kernel log to identify the issue that led to the crash.[‡] The manager stores the reports and the last crashing inputs for investigation and bug reporting.

2.5 Evaluation

We evaluated PeriScope and PeriFuzz by monitoring and fuzzing the communication between two popular Wi-Fi chipsets and their device drivers used in several Android smartphones.

2.5.1 Target Drivers

We chose Wi-Fi drivers as our evaluation target because they present a large attack surface, as evidenced by a recent series of fully remote exploits [34, 36]. Smartphones frequently connect to potentially untrusted Wi-Fi access points, and Wi-Fi drivers and peripherals implement vendor-specific, complex internal device-driver interaction protocols (*e.g.*, for offloading tasks) that rely heavily on DMA-based communication.

[‡] We used Syzkaller's report package to parse the kernel log.

The Wi-Fi peripheral chipset market for smartphones is dominated by two major vendors: Broadcom and Qualcomm. We tested two popular Android-based smartphones that each have a Wi-Fi chipset from one of these vendors, as shown in Table 2.3. We tested the Google Pixel 2, with Android 8.0.0 Oreo[§] and Qualcomm's `qcacld-3.0` Wi-Fi driver. We also tested the Samsung Galaxy S6, on which we installed LineageOS 14.1 and Broadcom's `bcmdhd4358` Wi-Fi driver. LineageOS 14.1 is a popular custom Android distribution that includes the exact same Broadcom driver as the official Android version for the Galaxy S6.

Note that although the Samsung Galaxy S6 has an IOMMU, it is not being used to protect the physical memory from rogue Wi-Fi peripherals. Regardless, we did conduct our experiments under the assumption that IOMMU protection *is* in place. Newer versions of the Samsung Galaxy phones do enable IOMMU protection for Wi-Fi peripherals.

2.5.2 Target Attack Surface

The code paths that are reachable from peripheral devices vary depending on the internal state of the driver (*e.g.*, is the driver connected, not connected, scanning for networks, *etc.*). In our evaluation, we assume that the driver has reached a steady state where it has established a stable connection with a network. We consider only the code paths reachable in this state as part of the attack surface. We analyzed this attack surface by counting (i) the number of allocation contexts that create attacker-accessible MMIO and DMA mappings and (ii) the number of driver code paths that are executed while the user is browsing the web.

Table 2.4 summarizes the MMIO and DMA allocation contexts in both device drivers, which create mappings that can be accessed by the attacker while the user is browsing the web. MMIO and DMA coherent mappings were established during the driver initial-

---

[§] `android-8.0.0_r0.28`

Table 2.4: The number of MMIO and DMA allocation contexts that create attacker-accessible mappings

Driver	MMIO	DMA Coherent	DMA Streaming
`qcacld-3.0`	1	9	5
`bcmdhd4358`	4	11	29

Table 2.5: The number of basic blocks executed under web browsing traffic per kernel control path. A basic block could run in interrupt context (**IRQ**), kernel thread or worker context (**Kernel Thread**), or others (**Others**). Some basic blocks can be reached in several contexts.

Driver	IRQ	Kernel Thread	Others	Hit / Instrumented
`qcacld-3.0`	1633 (36.9%)	2902 (65.6%)	672 (15.2%)	4427/81637
`bcmdhd4358`	743 (68.9%)	284 (26.3%)	301 (27.9%)	1078/23404

ization, and were still mapped to both the device and the driver by the time the user browses the web; DMA streaming mappings were destroyed after their use, but regularly get recreated and mapped to the device while browsing the web. Thus, an attacker on a compromised Wi-Fi chipset can easily access these mappings, and write malicious values in them to trigger and exploit vulnerabilities in the driver.

We then analyzed the code paths that get exercised under web browsing traffic, and classified these paths based on the context in which they are executed: interrupt context, kernel thread context, and other contexts (*e.g.*, system call context). Table 2.5 shows the results. Of all the basic blocks executed under web browsing traffic, 36.9% and 68.9% run in interrupt context for the `qcacld-3.0` and `bcmdhd4358` drivers, respectively. Some of the code that executes in interrupt context may not be reachable from any system calls through legal control-flow paths, and therefore may not be fuzzed by system call fuzzers.

Table 2.6: Allocation contexts selected for fuzzing. DC stands for DMA coherent, DS for DMA streaming, and MM for memory-mapped I/O.

Driver	Alloc. Context	Alloc. Type	Alloc. Size	Used For
`qcacld-3.0`	**QC1**	DC	8200	DMA buffer mgmt.
	QC2	DC	4	DMA buffer mgmt.
	QC3	DS	2112	FW-Driver message
	QC4	DS	2112	FW-Driver message
`bcmdhd4358`	**BC1**	DC	8192	FW-Driver RX info
	BC2	DC	16384	FW-Driver TX info
	BC3	DC	1536	FW-Driver ctrl. info
	BC4	MM	4194304	Ring ctrl. info

2.5.3 Target Mappings

We investigated how each of the active mappings are used by their respective drivers, and enabled fuzzing for DMA/MMIO regions that are accessed frequently, and that are used for low-level communication between the driver and the device firmware (*e.g.*, for shared ring buffer management). We used PeriScope to determine which regions the driver accesses frequently, and we manually investigated the driver's code to determine the purpose of each region.

For `qcacld-3.0`, we enabled fuzzing for two allocation contexts for DMA coherent buffers and two contexts for DMA streaming buffers. For `bcmdhd4358`, we enabled fuzzing for three allocation contexts for DMA coherent buffers and one allocation context for an MMIO buffer. Table 2.6 summarizes the allocation contexts for which we enable fuzzing; all the mappings allocated in those contexts are fuzzed.

Table 2.7: Unique device driver vulnerabilities found by PeriFuzz

Alloc. Context	Alloc. Type	Error Type	Analysis	Double-fetch	Status (Severity)	Impact
QC2	DC	Buffer Overflow	Unexpected RX queue index		CVE-2018-11902 (High)	Likely Exploitable
QC3	DS	Null-pointer Deref.	Unexpected message type		Confirmed (Low)[a]	DoS
QC3	DS	Buffer Overflow	Unexpected peer id		Known	Likely Exploitable
QC3	DS	Buffer Overflow	Unexpected number of flows		Known	Likely Exploitable
QC3	DS	Address Leak/Buffer Ovf.	Unexpected FW-provided pointer		CVE-2018-11947 (Med)[b]	Likely Exploitable
QC3	DS	Buffer Overflow	Unexpected TX descriptor id		Known	Likely Exploitable
QC4	DS	Reachable Assertion	Unexpected endpoint id		Known (Med)	DoS
QC4	DS	Reachable Assertion	Duplicate message		Known (Med)	DoS
QC4	DS	Reachable Assertion	Unexpected payload length		Known (Med)	DoS
BC1	DC	Buffer Overflow	Unexpected interface id	✓	CVE-2018-14852, SVE-2018-11784 (Low)	Likely Exploitable

[a] Qualcomm confirmed the vulnerability but they do not assign CVEs for low-severity ones.
[b] CVE assigned for the address leak.

2.5.4 Fuzzer Seed Generation

We used PeriScope's default tracing facilities to generate initial seed input files. For each selected allocation context, we first recorded all allocations of, and all read accesses to the memory mappings while generating web browsing traffic for five minutes. We then parsed the allocation/access log to generate unique seed input files. Finally, we used AFL's corpus minimization tool to minimize the input files. This tool replays each input file to collect coverage information and uses that information to exclude redundant files.

2.5.5 Vulnerabilities Discovered

Table 2.7 summarizes the vulnerabilities we discovered using our fuzzer. Each entry in the table is a unique vulnerability at a distinct source code location.

Disclosure We responsibly disclosed these vulnerabilities to the respective vendors. During this process, we were informed by Qualcomm that some of the bugs had recently been reported by external researchers or internal auditors. We marked these bugs as "Known". All the remaining bugs were previously unknown, and have been confirmed by the respective vendors. We included CVE numbers assigned to the bugs we reported. Also, we included the vendor-specific, internal severity ratings for these bugs if communicated by the respective vendors during the disclosure process.

Error Type and Impact Vulnerabilities found by PeriFuzz fall into four categories: buffer overflows, address leaks, reachable assertions, and null-pointer dereferences. We mark buffer overflows and address leaks as potentially exploitable, and reachable assertions and null-pointer dereferences as vulnerabilities that can cause a denial-of-service (DoS) attack by triggering device reboots.

Double-fetch Bugs We did not attempt to find double-fetch bugs in streaming DMA buffers, since we operated under the assumption that an IOMMU preventing such bugs is in place (see Section 2.3.1). That said, we did identify several double-fetch bugs in code that accesses coherent DMA buffers. These bugs can potentially be exploited, even when the system deploys an IOMMU. We discuss these bugs in detail in Section 2.5.7.

2.5.6 Case Study I: Design Bug in `qcacld-3.0`

One of the vulnerabilities we found in `qcacld-3.0` is in code that dereferences a firmware-provided pointer. PeriFuzz fuzzed the pointer value as it was read by the device driver. The driver then dereferenced the fuzzed pointer and crashed the kernel. An analysis of this vulnerability revealed that it is in fact a design issue. The pointer was originally provided by the driver to the device. Line 11 in Listing 1 turns a kernel virtual address, which points to a kernel memory region allocated at Line 4, into a 64-bit integer called `cookie`. The driver sends this `cookie` value to the device, thereby effectively leaking a kernel address.

An attacker that controls the peripheral processor can infer the kernel memory layout based on the cookie values passed by the driver. This address leak can facilitate exploitation of memory corruption vulnerabilities even if the kernel uses randomization-

```c
A_STATUS ol_txrx_fw_stats_get(...)
{
  ...
  non_volatile_req = qdf_mem_malloc(sizeof(*non_volatile_req));
  if (!non_volatile_req)
    return A_NO_MEMORY;

  ...

  /* use the non-volatile request object's address as the cookie */
  cookie = ol_txrx_stats_ptr_to_u64(non_volatile_req);

  ...
}
```

Listing 1: Kernel address leak in `qcacld-3.0`

based mitigations such as KASLR. This bug can be fixed by passing a randomly generated cookie value rather than a pointer to the device.

2.5.7 Case Study II: Double-fetch Bugs in `bcmdhd4358`

The `bcmdhd4358` driver contains several double-fetch bugs that allow an adversarial Wi-Fi chip to bypass an integrity check in the driver. Listing 2 shows how the driver accesses a coherent DMA buffer that holds meta-information about network data. At Line 4 and Line 5, the driver verifies the integrity of the data in the buffer by calculating and checking an XOR checksum. The driver then repeatedly accesses this coherent DMA buffer again. The problem here is that the device, if compromised, could modify the data between the point of the initial integrity check, and the subsequent accesses by the driver.

PeriFuzz was able to trigger multiple vulnerabilities by modifying the data read from this buffer *after* the integrity check was completed. We show one buffer overflow vulnerability in Listing 3, which was triggered by fuzzing the `ifidx` value used at Line 4. The overlapping fetch that occurred before this buffer overflow is a double-fetch bug, because the overlapping fetch can invalidate a previously passed buffer integrity check.

```
static uint8 BCMFASTPATH dhd_prot_d2h_sync_xorcsum(dhd_pub_t *dhd, msgbuf_ring_t *ring, volatile cmn_msg_hdr_t *msg, int msglen)
{
  ...
  prot_checksum = bcm_compute_xor32((volatile uint32 *)msg, num_words);
  if (prot_checksum == 0U) { /* checksum is OK */
    if (msg->epoch == ring_seqnum) {
      ring->seqnum++; /* next expected sequence number */
      goto dma_completed;
    }
  }
  ...
}
```

Listing 2: Initial fetch and integrity check in `bcmdhd4358`

```
void dhd_rx_frame(dhd_pub_t *dhdp, int ifidx, void *pktbuf, int numpkt, uint8 chan)
{
  ...
  ifp = dhd->iflist[ifidx];
  if (ifp == NULL) {
    DHD_ERROR(("%s: ifp is NULL. drop packet\n",
               __FUNCTION__));
    PKTFREE(dhdp->osh, pktbuf, FALSE);
    continue;
  }
  ...
}
```

Listing 3: Buffer overflow in `bcmdhd4358`

Thus, in addition to safeguarding the array access with a bounds check, the driver should copy the contents of the coherent DMA buffers to a location that cannot be accessed by the peripheral device, before checking the integrity of the data in the buffer. Subsequent uses of device-provided data should also read from the copy of the data, rather than the DMA buffer itself.

2.5.8 Case Study III: New Bug in `qcacld-3.0`

Listing 4 shows a null-pointer deference bug we discovered in the `qcacld-3.0` driver. The pointer to the `netbufs_ring` array dereferenced at Line 9 is null, unless the driver is configured to explicitly allocate this array. The driver configuration used by the Google Pixel 2 did not contain the entry necessary to allocate the array. Although the driver never executes the vulnerable code under normal conditions, we found that the vulnerable line *is* reachable through legal control flow paths.

```
1  static inline qdf_nbuf_t htt_rx_netbuf_pop(htt_pdev_handle pdev)
2  {
3    int idx;
4    qdf_nbuf_t msdu;
5
6    HTT_ASSERT1(htt_rx_ring_elems(pdev) != 0);
7
8    idx = pdev->rx_ring.sw_rd_idx.msdu_payld;
9    msdu = pdev->rx_ring.buf.netbufs_ring[idx];
10   ...
11 }
```

Listing 4: Null-pointer dereference in `qcacld-3.0`

It is difficult to detect this bug statically, as it requires a whole-program analysis of the device driver to determine if the `netbufs_ring` pointer is initialized whenever the vulnerable line can execute. PeriFuzz consistently triggered the bug, however. This vulnerability discovery therefore bolsters the argument that fuzzing can complement manual auditing and static analysis.

2.5.9 Performance Analysis

Page Fault

PeriScope incurs run-time overhead as it triggers a page fault for every instruction that accesses the monitored set of DMA/MMIO regions. We quantified this overhead

Table 2.8: Time consumed by PeriScope's page fault handler (measured in µ seconds)

	Mean	Minimum	Maximum
Tracing Only	117.6	99.8	194.5
Tracing + Fuzzing	227.8	182.7	379.7

by measuring the number of clock cycles spent inside PeriScope's page fault handler. We read the AArch64 counter-timer virtual count register `CNTVCT_EL0` when entering the handler and when exiting from the handler, and calculated the difference between the counter values, divided by the counter-timer frequency counter `CNTFRQ_EL0`. To minimize interference, we disabled hardware interrupts while executing our page fault handler. We also disabled dynamic frequency and voltage scaling.

We tested the page fault handler under two configurations. In one configuration, PeriScope calls the default pre- and post-instruction hooks that only trace and log memory accesses. In the other configuration, we registered PeriFuzz's instruction hooks to enable DMA/MMIO fuzzing. Table 2.8 shows the mean, minimum, and maximum values over samples of 500 page fault handler invocations for each configuration.

Note that we deliberately trade performance for deterministic, precise monitoring of device-driver interactions, by trapping every single access to a set of monitored mappings. In fact, this design allowed us to temporally distinguish accesses to the same memory locations, which was essential to find the double-fetch bugs. The drivers still function correctly, albeit more slowly, when executed under our system, making it possible to examine device-driver interactions dynamically and enabling PeriFuzz to fuzz it.

Fuzzing

PeriFuzz builds on PeriScope and has additional components that interact with each other, which incur additional costs. The primary contributors to this additional cost

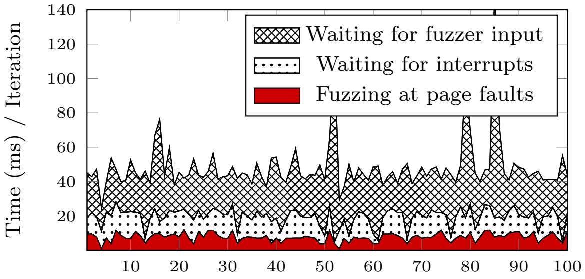
Table 2.9: Peak fuzzing throughput for each fuzzed allocation context

Driver	Alloc. Context	Peak Throughput (# of test inputs/sec)
`qcacld-3.0`	**QC1**	23.67
	QC2	15.64
	QC3	18.77
	QC4	7.63
`bcmdhd4358`	**BC1**	9.90
	BC2	14.28
	BC3	10.49
	BC4	15.92

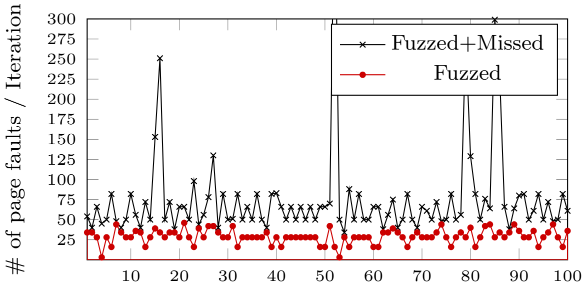
are: (i) waiting for the peripheral to signal the driver, (ii) waiting for a software interrupt to be scheduled by the Linux scheduling subsystem, (iii) interactions with the user-space fuzzer, which involve at least two user-kernel mode switches (i.e., one for delivering fuzzer inputs and the other for polling and retrieving feedback), and (iv) other system activities.

Peak Throughput We measured the overall fuzzing throughput to quantify the overhead incorporating all interactions between the PeriFuzz components. We only report the peak throughput in Table 2.9, since crashes and device driver lockups heavily impact the average fuzzing throughput (see Section 2.6.1). The inverse of the peak fuzzing throughput is a conservative lower bound for the execution time required to process a single fuzzer-generated input. Although we did not optimize PeriFuzz for throughput, we believe that these numbers are still in a range that makes PeriFuzz practical for dynamic analysis.

Overhead Breakdown To illustrate how the fuzzing throughput can be optimized, we present a breakdown of the fuzzing overhead. We divide each iteration of the fuzzing loop into three phases: (i) waiting for fuzzer input to be made available to our kernel module,



(a) The execution time of three phases



(b) The number of fuzzed and missed page faults

Figure 2.5: Fuzzing overhead breakdown

(ii) waiting for the device to raise an interrupt and for the driver to start processing it, and (iii) fuzzing the data read from monitored I/O mappings upon page faults. Once the driver has finished processing the interrupt, the next iteration begins. We measured the execution time of each phase in each iteration. To evaluate the impact of page faults on the fuzzing performance, we also counted the number of page faults triggered during each iteration.

We performed the experiment while fuzzing the buffer having the highest peak throughput (**QC1**). Figure 2.5a shows our measurements of per-phase execution time in a stacked manner, over 100 consecutive iterations of the fuzzing loop. 60% of the total execution time is spent on waiting for the next fuzzer input to be available. This delay is primarily caused by a large number of missed page faults, as hinted by Figure 2.5b. The current implementation of PeriFuzz can miss page faults, when they are triggered while PeriFuzz is preparing for the next input. This delay can be reduced by disabling page faults until the next input is ready. The delay caused by waiting for relevant interrupts, which accounts for 24.2% of the total execution time, can be reduced by forcing hardware to raise relevant interrupts more frequently.

The actual fuzzing at each page fault still takes 15.8% of the total execution time. One way to reduce this overhead is to trigger page faults only at first access to a monitored mapping within each iteration. At first access, the underlying page can be overwritten with the fuzzer input and then made present, so that subsequent accesses to the page within the same iteration do not trigger extra page faults. This would come, however, at the cost of precision, because it loses precise access tracing capability, effectively disabling overlapping fetch fuzzing as well as detection of potential double-fetch bugs.

2.6 Discussion

2.6.1 Limitations

We discuss problems that limit both the effectiveness and efficiency of PeriFuzz. These are well-known problems that also affect other kernel fuzzers, such as system call fuzzers.

System Crashes

The OS typically terminates user-space programs when they crash, and they can be restarted without much delay. Crashing a user-space program therefore has little impact on the throughput of fuzzing user-space programs. Crashes in kernel space, by contrast, cause a system reboot, which significantly lowers the throughput of any kernel fuzzer. This is particularly problematic if the fuzzer repeatedly hits shallow bugs, thereby choking the system without making meaningful progress. We circumvented this problem by disabling certain code paths that contain previously discovered shallow bugs. This does, however, somewhat reduce the effectiveness of our fuzzer as it cannot traverse the subpaths rooted at these blacklisted bugs. Note that this problem also affects other kernel fuzzers, *e.g.*, DIFUZE and Syzkaller [73, 74].

Driver Internal States

Due to the significant latency involved in system restarts, whole-system fuzzers typically fuzz the system without restarting it between fuzzing iterations. This can limit the effectiveness of such fuzzers, because the internal states of the target system persist across iterations. Changing internal states can also lead to instability in the coverage-guidance, as the same input can exercise different code paths depending on the system state. This means that coverage-guidance may not be fully effective. Worse, when changes to the persisting states accumulate, the device driver may eventually lock itself up. For example, we encountered a problem where, after feeding a certain number of invalid inputs to a driver, the driver decided to disconnect from the network, reaching an error state from which the driver could not recover without a device reboot. Existing device driver checkpointing and recovery mechanisms could be adapted to alleviate the problem [75, 76], because they provide mechanisms to roll drivers back to an earlier state. Such a roll back takes significantly less time than a full system reboot.

2.6.2 Augmenting the Fuzzing Engine

Although we used mutational, feedback-guided fuzzing to mutate the data stream on the device-driver interaction path, our fuzzing framework can also benefit from other fuzzing techniques. Like DIFUZE [73], static analysis can be introduced to infer the type of an I/O buffer, which can save fuzzing cycles by respecting the target type when mutating a value. The dependencies between device-driver interaction messages can also be inferred using static and trace analysis techniques [77, 78], which can help fuzzing stateful device-driver interaction protocols. Alternatively, developers can specify the format of an I/O buffer and/or interaction protocol in a domain-specific language [74, 79]. In addition to improving the mutation of the data stream, we could use system call fuzzers such as Syzkaller that generate different user-space programs [74]. These generated programs could actively send requests to the driver and potentially to the device, which in turn can increase reachable interrupt code paths. We believe that our modular framework allows for easy integration of these techniques.

2.6.3 Combining with Dynamic Analysis

Our framework runs in a concrete execution environment; thus, existing dynamic analysis tools can be used to uncover silent bugs. For example, kernel sanitizers such as address sanitizer and undefined behavior sanitizer can complement our fuzzer [80, 81]. Memory safety bugs often silently corrupt memory without crashing the kernel. Our fuzzer, by itself, would not be able to reveal such bugs. When combined with a sanitizer, however, these bugs *would* be detected. Other dynamic analysis techniques such as dynamic taint tracking can also be adapted to detect security-critical semantic bugs such as passing security-sensitive values (*e.g.*, kernel virtual addresses) to untrusted peripherals.

2.7 Related Work

2.7.1 Protection against Peripheral Attacks

An IOMMU isolates peripherals from the main processor by limiting access to physical memory to regions configured by the OS. Markuze et al. proposed mechanisms that can achieve strong IOMMU protection at an affordable performance cost [65, 66]. Several other work proposed mechanisms that can limit functionalities exposed to potentially malicious devices [82–84]. Cinch encapsulates devices as network endpoints [83], and USBFILTER hooks USB APIs [84], to enable user-configurable, fine-grained access control. However, neither IOMMU protection nor fine-grained access control prevents exploitation of vulnerabilities found in code paths that are still reachable from the device.

The effects of vulnerabilities on these valid code paths can be mitigated by isolating device drivers from the kernel [85–88]. Android, for example, switched from the kernel-space Bluetooth protocol stack [89] to a user-space Bluetooth stack [90]. The OS kernel merely acts as a data path by forwarding incoming packets to the user-space Bluetooth daemon process. This approach can mitigate vulnerabilities in the device driver because the driver cannot access kernel memory and cannot execute privileged instructions. The daemon process still runs at a higher privilege level than standard user-space processes, however, and therefore remains an attractive target for adversaries looking to access sensitive data [91]. Additionally, this approach is currently not viable for certain types of device drivers. High-bandwidth communication devices such as Wi-Fi chips, for example, cannot afford the mode and context switching overhead incurred by user-space drivers.

2.7.2 Kernel Fuzzing

Most kernel fuzzing tools focus on the system call boundary [67, 73, 74, 77, 78, 92–96]. DIFUZE uses static analysis and performs type-aware fuzzing of the IOCTL interface, which can expose a substantial amount of driver functionality to user space [73]. Syzkaller, a coverage-guided fuzzer, fuzzes a broader set of system calls, based on system call description written in a domain-specific language [74]. IMF infers value-dependence and order-dependence between system call arguments by analyzing system call traces [77]. kAFL uses Intel Processor Trace as a feedback mechanism, to enable OS-independent fuzzing [67]. Digtool uses virtualization to capture and analyze the dynamic behavior of kernel execution [92].

PeriFuzz can be augmented with techniques that facilitate type-aware fuzzing [73, 74, 77, 78], as discussed in Section 2.6.2. Tools based on certain hardware features can fuzz closed-source OSes [67, 92], but smartphones often do not contain or expose the necessary hardware features to the end user. For example, most smartphone OSes block access to the bootloader and to hypervisor mode, thus preventing end users from running code at the highest privilege level [97]. None of these fuzzers target DMA/MMIO-based interactions between drivers and devices, nor do they cover code paths that are not reachable from system calls (*e.g.*, interrupt handlers).

2.7.3 Kernel Tracing

There are many general-purpose tools to monitor events in the Linux kernel. Static kernel instrumentation mechanisms such as Tracepoint allow the developer to insert so-called probes [98]. Ftrace and Kprobe are dynamic mechanisms that can be used to probe functions or individual instructions [99, 100]. eBPF, the extended version of the Berkeley Packet Filter mechanism, can attach itself to existing Kprobe and Tracepoint probes for further processing [101]. LTTng, SystemTap, Ktap and Dprobe are higher level primitives that build on the aforementioned tools [102–105].

These tools, however, are not well suited to monitoring device-driver interactions, because they require developers to identify and instrument each device-driver interaction. These manual efforts can be alleviated by using page fault based monitoring, which Mmiotrace uses to trace MMIO-based interactions in x86 and x86-64 [106]. However, Mmiotrace does not support the DMA interface, *i.e.*, DMA coherent and streaming buffers, and it lacks the ability to manipulate device-driver interactions. In contrast, PeriScope can trace both MMIO and DMA interfaces, and can be used to manipulate device-driver interactions by plugging in PeriFuzz, enabling adversarial analysis of device drivers.

2.7.4 Kernel Static Analysis

Static analysis tools can detect various types of kernel and driver vulnerabilities [40, 107–110]. Dr. Checker runs pointer and taint analyses specifically tailored to device drivers, and feeds the analysis results to various vulnerability detectors [109]. K-Miner uses an inter-procedural, context-sensitive pointer analysis to find memory corruption vulnerabilities reachable from system calls [110]. Symbolic execution can complement these static analyses to work around precision issues. Deadline [40], for example, uses static analysis to find multi-reads in the kernel, and symbolically checks whether each multi-read satisfies the constraints to be a double-fetch bug. With the help of this symbolic checking, Deadline can precisely discern double-fetch bugs from statically identified multi-reads. Generally speaking, however, techniques based on symbolic execution may not scale well due to the path explosion problem.

Static analysis techniques have traditionally been applied to the system call interface only. Although the core ideas can apply to the hardware-OS interface too, statically identifying the necessary entry points may not be as trivial as with system calls, since accesses to an I/O mapping are difficult to distinguish from other memory accesses, and interrupt processing code can run in different, unrelated contexts (e.g., software interrupt context, kernel thread context, etc.).

2.7.5 Finding Double-fetch Bugs

Double-fetch bugs are a special case of time-of-check-to-time-of-use (TOCTTOU) race conditions. They occur when privileged code fetches a value from a memory location multiple times, while less privileged code is able to change the value between the fetches [41, 42]. Previous work explored multiple reads of user-space memory from OS kernels or from trusted execution environments [40, 42, 92, 108, 111], and multiple reads of memory shared between different hypervisor domains [112]. They either use static analysis (e.g., static code pattern matching [108] and symbolic execution [40]), or dynamic analysis (e.g., memory access tracing followed by pattern analysis [42, 92, 112] and cache behavior-guided fuzzing [111]). PeriFuzz is also a dynamic approach, but targets a different attack surface: I/O memory mappings shared between peripheral devices and kernel drivers.

PeriFuzz and DECAF are currently the only two tools that are sufficiently generic to support double-fetch fuzzing without instrumentation or manual analysis of the target code [111]. DECAF cannot fuzz double-fetches from MMIO and DMA coherent mappings, however, because these mappings are typically uncached, and DECAF relies on cache side channels to detect double-fetches.

2.8 Conclusion

In this chapter, we discuss the remote attack vectors of the OS kernel, and the potential disastrous consequences of a remote kernel compromise. While previous security efforts have focused on securing the system call boundary, recent exploits have demonstrated alternative paths to kernel compromise, such as through compromised peripheral devices. We introduces PeriScope, a probing framework based on the Linux kernel, designed to analyze device-driver interactions. It can passively monitor and log traffic, or actively mutate data streams using the PeriFuzz fuzzing component. The framework has been evaluated on Wi-Fi drivers of two popular vendors, uncovering 15 vulnerabilities, including 9 previously unknown ones. This work emphasizes the importance of addressing vulnerabilities along the hardware-OS boundary, and highlights the need for a framework that can identify and mitigate such risks. The PeriScope framework fills that void.

Chapter 3

Understanding Security Issues in the NFT Ecosystem

A *Non-Fungible Token* (NFT) is an ownership record stored on a blockchain (such as the Ethereum blockchain). While digital items, such as pictures and videos, are the most common assets traded as NFTs, the sale of physical assets, *e.g.*, postal stamps [113, 114], gold [115], real estate [116], physical artwork [117], *etc.*, is also steadily gaining popularity. In the cryptocurrency world, an NFT is the equivalent of a conventional proof-of-purchase, such as a paper invoice or an electronic receipt. Among other things, what make NFTs attractive are *verifiability* and *trustless transfer* [118]. Verifiability means that sales are recorded as blockchain transactions, which makes tracking of ownership possible. In addition, the NFT concept allows for the trading of digital assets between two mutually distrusting parties, as both the crypto payment and the asset transfer happen atomically in a single transaction.

Legitimacy is one of the big issues with NFTs, as nothing prevents an impostor from "tokenizing" and selling someone else's art, while the creator remains oblivious of the fraud. With the current state of affairs, the onus of verifying the token is on the buyer.

Unfortunately, this is not always easy. For instance, in August 2021, a perpetrator impersonated the popular British graffiti artist Banksy and sold an NFT [119] that featured a “fake” art piece by the artist for $336K USD through an online auction. While NFTMs try to thwart such attacks by mandating account validation, typically through an artist’s social media presence, another scammer punched a hole through RARIBLE’s verification process and managed to get a fake account associated with the renowned artist Derek Laufman verified [120]. Counterfeits NFTs, also called *copycats* or *parody* projects, resemble reputable collections and purport to have been created by reputable sources. For example, the early NFT project CRYPTOPUNKS has numerous clones, such as CRYPTO**Ph**UNKS. In some scenarios, scammers set up unauthorized customer support channels and social media accounts that pretend to be affiliated with NFTMs in an effort to steal customer information and compromise accounts [121]. Also, there is evidence of *rug-pulls*, where the owner/creator of an NFT unscrupulously hypes an asset in order to inflate its value, only to cash out, leaving others to suffer from the subsequent decline in value. One such example is the ETERNAL BEINGS collection, which was promoted by the popular American rapper Lil Uzi Vert through his TWITTER account with 8.5M followers. Soon after the initial investment by the buyers, he deleted all of his tweets, causing the token values to plummet [122].

To the best of our knowledge, however, the existing literature has not explored the security challenges in the emerging NFT ecosystem, or performed a systematic and comprehensive analysis of the associated threats. Our work fills that void. First, we identify three components constituting the NFT ecosystem. We then analyze each component to discover security, privacy, and usability issues, as well as economic threats. We hope that our work will be helpful both for NFT marketplaces and their users. We envision this work as a guide to help NFTMs to avoid mistakes while making users aware of the perils of the NFT space.

3.1 Background

In this section, we introduce the building blocks of the Ethereum ecosystem, with an emphasis on non-fungible tokens (NFTs) and the economy that has grown around them.

The Ethereum Blockchain. Ethereum is the technology powering the cryptocurrency Ether (ETH) and thousands of decentralized applications (dApps). The Ethereum blockchain is a distributed, public ledger where transactions are mined into *blocks* by *miners* who solve cryptographic Proof of Work (PoW) challenges. In this ecosystem, an *account* is an entity represented by an address that is capable of submitting transactions. There are two types of accounts in Ethereum: externally owned accounts (EOA), which are controlled by anyone holding the corresponding private key, and contract accounts, which contain executable pieces of code, called smart contracts. A smart contract is a program run by the Ethereum Virtual Machine (EVM), which leverages the blockchain to store its persistent state. A *transaction* is the transfer of funds between accounts, or an invocation of a contract's public method. The address that sends the funds or interacts with the contract is denoted by `msg.sender`.

Non-Fungible Token (NFT). In the real world, tokens are representations of facts, such as the position in a queue or the authorization to access a facility. In Ethereum, tokens are digital assets built on top of the blockchain. Unlike Ether, which is the native (built-in) cryptocurrency of the Ethereum blockchain, tokens are implemented by specialized smart contracts. There are two main types of tokens: fungible and non-fungible. All the copies of a *fungible* token, usually conforming to the ERC-20 interface [123], are identical and interchangeable. Such tokens can act as a secondary currency within the ecosystem, or can represent someone's stake in an investment. On the other hand, all the copies of *non-fungible* tokens, usually conforming to the ERC-721 [124] interface, are unique, and each token represents someone's ownership of a specific digital asset, such

```
setApprovalForAll(address _operator,
                  bool _approved) external
approve (address _approved,
         uint256 _tokenId) external payable
transferFrom (address _from, address _to,
              uint256 tokenId) external payable
tokenURI (uint256 _tokenId)
          external view returns (string)
```

Figure 3.1: Important methods defined in ERC-721.

as ENS domains [125] and CryptoKitties [126], or a physical asset, like a gold bar.

ERC-721 [124] is by far the most popular standard for implementing non-fungible tokens on Ethereum. The standard interface defines a set of mandatory and optional API methods that a token contract needs to implement. Figure 3.1 presents a few of those API methods relevant to our discussion.

Each NFT has its own ID (to keep track of these unique tokens), which is referred to as `_tokenId`. In ERC-721, an *operator* is an entity that can manage all of an NFT owner's assets. In other words, an NFT owner can delegate the authority to act on her assets to an operator. Depending on whether the `_approved` argument is set, the `setApprovalForAll()` method either adds or removes the address `_operator` from/to the set of the operators authorized by the `msg.sender` (the NFT's owner). Unlike an operator, who can operate on all the assets of an owner, ERC-721 defines a *controller* as an entity who is authorized to operate on one single asset held by an owner. The `approve()` method approves the address `_approved` as the controller of the asset `_tokenId`. An operator, a controller, or the owner can call the `transferFrom()` method to transfer the token `_tokenId` from the current owner's `_from` address to the `_to` address.

When an NFT is created (minted), the creator can optionally associate a URL with the NFT. That URL, called `metadata_url`, should point to a JSON file that conforms to the ERC-721 Metadata JSON Schema [124]. The JSON file stores the details of the asset,

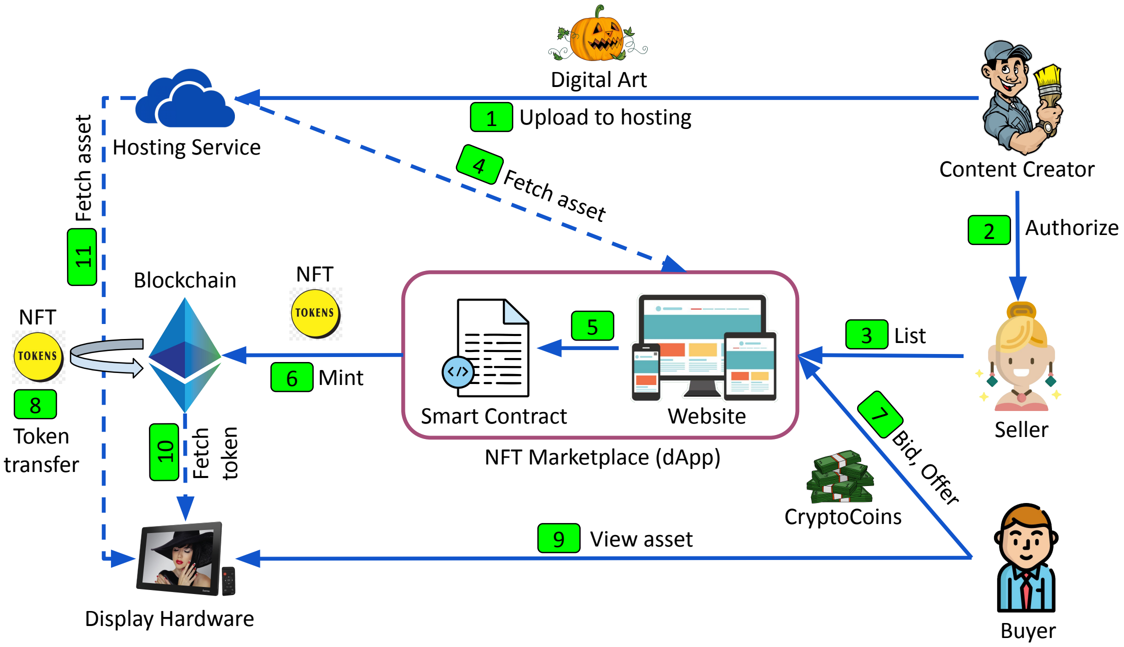


Figure 3.2: Anatomy of the NFT ecosystem showing all the marketplace actors, and their mutual interaction. The dotted and solid lines indicate data-centric and command-centric communication channels, respectively.

e.g., its `name` and `description`, and also contains an `image` field storing a URL, called `image_url`, that points to the asset. In this way, an NFT essentially connects an asset with the record of its ownership. Given a `_tokenId`, the associated `metadata_url` can be retrieved by querying the `tokenURI()` API of the contract. Interestingly, the creation and destruction of NFTs ("minting" and "burning") are not a part of the standard. Typically, `mint()` is defined as a public function restricted to the contract creator, and invoked by passing `metadata_url` as an argument. Minting can also be done during contract creation by calling `mint()` through the contract's constructor.

InterPlanetary File System (IPFS). IPFS [127] is a distributed, peer-to-peer, permissionless file system. Anyone can join the IPFS overlay network. A data item d is assigned a unique immutable address, also known as content identifier (CID): $cid = H(d)$, which is the hash H of the file's content d. Therefore, when the content of the file changes, the CID changes as well. The content of a file is first split into blocks. All the storage elements, *i.e.*, a directory, the files inside the directory, and the blocks within those files, are stored in a directed acyclic graph structure called a Merkle DAG. IPFS maintains a distributed hash table (DHT) split across all the nodes in the network to store provider records, which locate those peers that store the requested content. To retrieve a data item d, a node first looks up the providers $P(d)$ in the DHT, and then requests d from the members of $P(d)$.

3.2 Anatomy of the NFT Ecosystem

In this section, we provide an overview (Figure 3.2) of the economy that has developed around NFTs. Specifically, we identify the actors that participate in the ecosystem and the components they interact with.

Users. NFTs are often used to sell digital collectibles and artwork, *e.g.*, images, audio files, and videos. The users in the NFT ecosystem belong to one of three categories: *content creator*, *seller*, and *buyer*. First, the creators create digital content and upload it 1 to hosting services (an *external entity*) to make the art publicly available. When it comes to selling the content, some creators are not technical enough to turn their art into an NFT, and put it as a token on the blockchain. Therefore, they authorize 2 sellers to mint NFTs 6 and offer it on marketplaces. In other cases, a content creator is also taking the role of the seller. Once listed on a marketplace 3, buyers can buy the artwork at a listed price, make offers, or place bids 7. If their offer is accepted or they win an auction, the NFT is transferred 8 by invoking the `transferFrom()` API (Section 4.1) from the seller to the buyer to reflect the change in ownership.

Marketplaces. NFT marketplaces (NFTM) are dApp platforms where NFTs (also referred to as *assets*) are traded. There are typically two main components of an NFTM—

a user-facing web frontend, and a collection of smart contracts that interact with the blockchain. Users interact with the web app, which, in turn, sends transactions to the smart contracts on their behalf [5]. Primarily, there are two types of contracts: **(i)** marketplace contracts, which implement the part of the NFTM protocol that interacts with the blockchain, and **(ii)** token contracts, which manage NFTs. Marketplaces typically allow users to perform the following activities: **(a)** user authentication, **(b)** token minting, **(c)** token listing, and **(d)** token trading. The token-related activities are collectively called *events*. Depending on where these events are stored, three broad types of NFTM protocol design are possible: **(i)** *on-chain*: all the events live on the blockchain. Since every action costs gas, this design makes the NFTM operationally expensive for the users. NFTMs that follow this design include Axie, CryptoPunks, Foundation, and SuperRare. **(ii)** *off-chain*: the events are recorded in a centralized, off-chain database managed by the NFTM. Users perform various activities by interacting with the web app, not the blockchain, and, therefore, this design is gas-friendly. Nifty is an example of an off-chain NFTM. **(iii)** *hybrid*: depending on their type, events are stored either on-chain or off-chain. To ensure the integrity of the operation, on-chain and off-chain events are tied together with a cryptographic check. OpenSea and Rarible follow this model.

▶ **User authentication.** Users first need to register with the NFTMs to access their services. Post-registration, two different authentication workflows are possible: **(a)** classic credentials-based (username/password), or **(b)** signature-based. With the latter, the user is first asked to sign a challenge string. Then, the marketplace recovers [128] the address of the signer (user) from the elliptic-curve signature. OpenSea, Rarible, Foundation, CryptoPunks, and SuperRare follow this model. Since Ethereum private keys are essentially unguessable [129], this authentication method is generally more secure than traditional passwords (passwords are typically drawn from a limited set of characters, shorter in length, and easier to brute-force).

▶ **Token minting.** A token is *minted* (created) 6 by calling the appropriate method of the token contract, which generally complies with the ERC-721 or ERC-1155 standard. A single token contract can manage the ownership of a number of NFTs. Every NFT is assigned an integer called `_tokenId`. Therefore, an NFT is uniquely identified by the ⟨`token_contract_address`, `_tokenId`⟩ pair on the blockchain. A "family" of NFTs, which are either similar, or based on a common theme, called a *collection*, *e.g.*, CryptoPunks. An NFT can be minted in many different ways: **(a)** *default contract*: the token is minted as part of a pre-deployed, designated token contract managed by the marketplace. NFTMs like OpenSea, Foundation, SuperRare, *etc.*, provide a default contract to hold NFTs when no custom contract is deployed by the creator. **(b)** *replica contract*: the NFTM itself deploys a contract on behalf of the creator to manage the collection that the NFT is a part of. Deployed contracts have identical bytecode, but are customized through initialization parameters. Examples of such NFTMs includes Nifty and Rarible. Since both *default* and *replica* contracts are managed by the NFTM, together they are called *internal* token contracts. **(c)** *external contract*: the creator independently deploys a custom contract to manage the collection, and later imports it to the marketplace. To be interoperable with the NFTMs, external contracts must follow a well-established token standard. Otherwise, a custom integration is needed. OpenSea and Rarible allow external contracts on their platforms. A single token contract can manage one or more collection. Typically, *replica* or *external* contracts manage a single collection, while the marketplace *default* contract manages several. In the latter case, the NFTM dApp maintains an off-chain association between the set of `_tokenId`s and the collection those belong to.

▶ **Token listing.** Once created, a seller lists their assets for sale 3. To list an NFT on a platform, some NFTMs, *e.g.*, Foundation, SuperRare, Nifty, mandate either the seller or the entire collection (that the NFT is a part of) to be verified. Even for the NFTMs where verification is optional, for example, OpenSea, Rarible, getting an artist or a collection verified provides credibility and increases buyers' confidence. NFTMs display special badges on verified profiles of artists and collections, which helps in building a brand, and receive preferential treatment to boost sales – such as search priority and safe-listing to suppress safety-related alerts before the purchase.

▶ **Token trading.** Buyers can make offers, or place bids 7 on the assets on sale. When an offer is accepted, or an auction is settled, the NFTM transfers 8 assets from the seller's account to the buyer's. Usually, this is when the NFTMs charge a fee for the service they offer. A few key aspects of the NFTM bidding system are discussed below: **(i)** *Pricing protocol*: The bid price can either increase or decrease with every bid. In an English auction, the bid opens at a reserve price, which is the minimum price the seller is willing to accept for an NFT. Subsequent bids from the buyer gradually increase the price. The NFT goes to the highest bidder. The English auction approach is used by most NFTMs, *e.g.*, OpenSea, Foundation, and SuperRare. In a Dutch auction, the bid opens at a high price. Subsequent bids from the seller gradually decrease the price. The NFT goes to the bidder who first accepts a bid. Axie follows the Dutch auction pattern. **(ii)** *Bid storage*: Bids can be stored either on-chain, *e.g.*, CryptoPunks, Foundation, SuperRare, or off-chain, *e.g.*, Nifty, Rarible, OpenSea. There are protocols, such as Wyvern used by OpenSea, which keep both the sell order (listing) and the bids off-chain for gas efficiency, though the order matching and the NFT transfer happen on-chain. Therefore, the marketplace contract cryptographically verifies the buy order against the associated sell order to prevent a malicious buyer from either buying an item that is not on sale, or tampering with an existing sell order. **(iii)** *Active bids*: Some NFTMs disallow multiple active bids on the same asset. For example, in CryptoPunks, Foundation, or SuperRare, when a bidder outbids the current top bidder, the latter gets automatically refunded. **(iv)** *Bid withdrawal*: Some NFTMs, such as CryptoPunks, allow the withdrawal of bids, while others, for example, Foundation, do not. **(v)** *Bid settlement*: Bid settlement does not require seller's intervention in most cases, *i.e.*, the asset automatically goes to the highest bidder. However, for some NFTMs like CryptoPunks, the bid has to be explicitly accepted by the seller.

When an item is sold by a seller other than the creator, it is called a secondary sale. Royalty is the payment made to the creator for every such secondary sale. Before the first (primary) sale takes place, the creator specifies the royalty amount, which is then deducted from every secondary sales and given to the creator. The deduction happens either **(i)** on-chain, where royalty is calculated by the marketplace contract during the *buy* transaction, or **(ii)** off-chain, where the NFTM dApp keeps track of the royalty accumulated from all the sales.

External entities. External to both NFTMs and blockchain, there are services and devices that provide the necessary infrastructure for the system to work. For example, creators store 1 their artwork on web servers or storage services such as Amazon S3 or IPFS. When buyers purchase the NFT, they can exercise their *bragging right* by displaying the art on photobook-style websites or digital NFT photo-frames. The websites, photo-frames 11, and NFTMs 4 fetch tokens from the blockchain 10, and respective artwork from those services.

3.3 Analysis approach

This section studies scams, malpractice, and security issues in the NFT ecosystem. In particular, we investigate the following research questions related to the three entities identified in the previous section, *i.e.*, users, marketplaces, and external entities: **(RQ1.)**

Marketplace	Trading Volume	Assets	Events	Asset Collection	Event Collection
OpenSea [130]	4.32B	12,215,650	349,911,634	A, B	A
Axie [131]	1.75B	891,238	487,486	B	B
CryptoPunks [132]	1.18B	9,999	172,157	B	B
Rarible [133]	199.42M	72,509	1,864,997	B	B, W
SuperRare [134]	106.87M	28,676	198,848	B	B
Sorare [135]	97.42M	298,219	1,392,292	B	W, B
Foundation [136]	68.19M	112,120	508,349	B	B
Nifty [137]	300.12M	-	-	-	-

Table 3.1: Characteristics of the marketplace dataset. A: API access, W: Web scraping, B: Blockchain parsing.

Are there weaknesses in the way NFTMs operate today, and can those be exploited (Section 3.4)? **(RQ2.)** How and to what extent do external entities pose a threat to the NFT ecosystem (Section 3.5)? **(RQ3.)** Are users involved in any fraud or malpractice resulting in the financial loss for others (Section 3.6)?

We used a hybrid (both qualitative and quantitative) approach to answer RQ1, and a quantitative approach for both RQ2 and RQ3. The rest of this section discusses how we collected the data for the quantitative analysis, and we provide a rationale for choosing the specific NFT marketplaces that we examined in more detail.

Marketplace selection. In line with previous work [138], we use DappRadar [4], a popular tracker of dApps, to select the most relevant marketplaces. We selected 8 out of a total of 35 marketplaces (Table 3.1) listed in DappRadar. This selection was based on the following two criteria: **(a)** backed by the Ethereum blockchain, and **(b)** the "all-time" trading volume is over 50M USD as of June 15, 2021.

Data collection. We collect two different types of data: **(a)** information about the NFTs (assets), *e.g.*, collection name, asset URI, metadata URI, *etc.*, traded on the different marketplaces, **(b)** NFT-related events, such as mint, buy, sell, auction creation, placing of a bid, acceptance of a bid, transfer, *etc.*, generated as a result of marketplace activity. We provide the details of the collected data in Table 3.4. To conduct differential analysis for one of the studies, we needed to monitor how the details of certain assets change over a period of time. Therefore, we crawled the same set of assets three times with a three-month interval between two subsequent crawls: in June 2021, September 2021, and finally in December 2021. Moreover, we collected event information continuously between the June and September crawls. We use COINGECKO [139] API to fetch historical prices of the cryptocoins to convert the pricing information to their equivalent USD value.

Asset and event information: The first step to collect asset and event information is *asset enumeration*, *i.e.*, obtaining the list of assets traded on a marketplace. Once enumerated, we collect the asset and event information for those assets. For both the steps, we employ three different strategies, subject to marketplace restrictions:

1) API access: If a marketplace exposes an appropriate API, we use it to retrieve the list of assets and events. Unfortunately, the APIs are often record-limited, *e.g.*, for a specific query, OPENSEA's API returns at most $10,000$ assets. However, the total number of assets listed on their website was 18.2M at the time of crawling. As a workaround, we generate API requests with combinations of `sort` and `filter` parameters to fetch different sets of assets with every request.

2) Web scraping: If a marketplace does not provide an API interface, but its terms and conditions (T&C) do not disallow scraping of their web interface, we crawl the assets and events data from the website.

3) Blockchain parsing: If a marketplace neither provides an API nor allows web scraping, we retrieve asset and event data directly from the blockchain, if possible. Trading activities of a decentralized marketplace are handled by smart contracts that are well-known. Leveraging the ABI (Application Binary Interface) of the contracts published in ETHERSCAN, we parse historical transactions, *e.g.*, `atomicMatch()` in case of OPENSEA, to retrieve asset and event details.

Our asset collection is best-effort, as it is impossible to enumerate all the listed assets in a marketplace. This is due to various reasons mentioned above, such as the absence of marketplace APIs, their rate limits, and T&C prohibiting any crawling activity. Table 3.1 shows the number of assets and events collected for each marketplace, and the strategies used to collect data. Since NIFTY does not provide an API, prohibits web scraping through T&C, and stores events off-chain, we were unable to collect data on the marketplace activities.

Measurement study. We utilize the asset and event data we collected to perform several measurement studies, which are described in the subsequent sections. We would like to emphasize that we attain reasonable coverage, *e.g.*, OPENSEA, the largest NFTM that accounts for 89.63% of assets in our dataset, listed 18.2M assets in their website at the time of crawling. We crawled 12.2M assets, which is 66.94% of the size of the marketplace. Since OPENSEA contributes to the most number of assets in our dataset, we use only OPENSEA in Section 3.4 (unless the study requires cross-NFTM analysis) and Section 3.5, as only that dataset would be representative enough to capture the extent of the issues we quantified in those sections. However, since we measured the occurrences of trading malpractices per NFTM in Section 3.6, we used assets from all the marketplaces. We provide the Ethereum addresses of the major contracts relevant to our study in Appendix 3.12.

3.4 Issues in NFT Marketplaces

In this section, we identify weaknesses in the design of NFTMs, which, when abused, pose a significant risk in the form of financial loss to both the marketplaces and its users. For this part of the study, we gathered information from public security incidents, attacks, and abuses reported on various blogs and technical reports, direct interactions with individual marketplaces, and marketplace documentation. We have systematized our findings

Issues	**Marketplaces**							
	OpenSea	Axie	CryptoPunks	Rarible	SuperRare	Sorare	Foundation	Nifty
User authentication								
U1. Identity verification	✗	✗	✗	✗	✗	✗	✗	✗
U2. Two-factor authentication	N	✗	N	N	N	O	N	✓
Token minting								
M1. Verifiability of token contracts	✗	N	N	✗	N	N	N	✗
M2. Tampering token metadata								
M2.1 Changing metadata url	P	✓	✗	✗	P	P	✗	✗
M2.2 Decentralized metadata	O	✗	✗	O	O	✗	M	O
Token listing								
L1. Principle of least privilege	✓	✓	✓	✓	P	✓	✗	✗
L2. Invalid caching	✓	N	N	✓	N	N	N	N
L3. Seller / collection verification	O	N	N	O	M	N	M	M
Token trading								
T1. Lack of transparency	✗	✗	✗	✗	✗	✗	✗	✓
T2. Fairness in bidding	✗	✓	✓	✗	✓	✗	✓	✗
T3. Royalty and fee evasion								
T3.1 Cross-platform	✗	✗	N	✗	✗	✗	P	✗
T3.2 Post-sales modification	✓	✗	N	✓	✗	✗	✗	✗

Table 3.2: Issues in the NFT marketplaces. O: Optional, M : Mandatory, P: Partial, N: Not applicable, ✓: Exists, ✗: Does not exist.

by connecting those issues with the marketplace activities discussed in Section 3.2, and then quantified, whenever possible, the prevalence/impact of those issues. Lastly, we systematically evaluated the existence of each of the issues across all the marketplaces (Table 3.2).

3.4.1 User Authentication

(U1) Identity verification. Art in the physical world has been used in money laundering schemes [140]. NFTs might make this process easier, as trades are executed by anonymous users, and there are no physical artworks to be transported. Identity verification is the first step to deter such criminals. Major crypto exchanges, such as Coinbase and Binance US, are highly regulated. To create an account with these exchanges, one needs to provide personally identifiable information (PII), *e.g.*, name, residential address, social security number (SSN), along with supporting documents confirming these details. Without getting the identity verified, it is either impossible to use the platform, or it can only be used with tight financial restrictions in place. To investigate if the NFTMs impose similar regulatory restrictions, we interacted with them by creating accounts. We discovered that no NFTM has made any steps towards enforcing KYC (Know Your Customer) rules nor implemented AML/CFT (Anti-Money Laundering/Combating the Financing of Terrorism) measures. As a result, apart from being able to hide the identity, a user can create several accounts on the platform that are hard to be traced back to one single entity.

(U2) Two-factor authentication. Enabling 2FA (Two-Factor Authentication) greatly enhances the security of a password-based authentication workflow. While traditional financial institutions like banks, brokerages, and cryptocurrency exchanges, such as COINBASE and BINANCE, provide 2FA as an option, it is not yet a ubiquitous option for NFTMs. SORARE manages a user's wallet on her behalf. As a result, an attacker who is able to login into an account can download the user's Ethereum private key associated with the wallet, and transact on behalf of her. Though SORARE does support 2FA, it is not enabled by default. 2FA was also optional for NIFTY users until the infamous hack [141] that compromised a number of accounts in March 2021. According to their initial assessment, none of the impacted accounts used 2FA when the hack took place.

3.4.2 Token Minting

(M1) Verifiability of token contracts. A token contract is considered "verifiable" if its source code is submitted to ETHERSCAN. Given the functional complexity of these token contracts, source code is much easier to audit than bytecode. Verifiability of external token contracts is crucial as they can be malicious or buggy. As an example, OPENSEA users complained about a malicious token contract that did not transfer tokens after purchase. Also, to make a particular NFT valuable, sometimes NFT projects promise to circulate only a certain number (rarity) of that token. A malicious token contract can be abused to mint more tokens than the *rarity* threshold, thus dropping the token's price, which hurts the buyers. A malfunctioning contract can burn gas without even doing any real work, *e.g.*, almost all `Purchase` events of the `CelebrityBreeder` contract failed with errors. Ideally, an NFT project should make the source of the underlying token contract available for public scrutiny before the NFTs are minted to make sure that they are neither malicious nor buggy. Unfortunately, none of the NFTMs that support external token contracts mandates such contracts to be open-source.

▶ **Quantitative analysis.** To enumerate how abundant closed-source NFT tokens are, we queried ETHERSCAN API for every token contract in our dataset to check if its source is present. Out of 11,339 token contracts, 8,122 (71.63%) were open-source, while the remaining 3,217 (28.37%) were closed-source, of which 7,850 (96.65%) and 3,209 (99.75%) tokens belong to OPENSEA, respectively.

Further, we intended to evaluate if closed-source tokens are more likely to exhibit malicious behavior than open-source ones. Since NFTMs take down NFTs when they observe or receive a report of either an abuse or a violation of the T&C, we consider "takedown" as an indirect (yet strong) indication of a token being found malicious. According to our observation, 1,765 (55.00%) closed-source tokens were taken down by OPENSEA between June and December, which account for $328.8M USD in trading volume. On the contrary, only 606 (7.72%) open-source tokens were taken down during the same span.

(M2) Tampering with token metadata. The metadata of a token holds the pointer to the corresponding asset. Hence, if the metadata changes, the token loses its significance. The ERC-721 standard for NFTs actually allows for the possibility to change a token's metadata. However, when an NFT represents a particular asset (such as a piece of art) that is sold, changing the metadata violates the expectation of the buyer. The location and the content of the metadata are decided at the time of minting. A malicious creator/owner $\mathcal{A}$ can alter the metadata by manipulating either of the two post-minting: **(i)** by changing the `metadata_url`, and **(ii)** by modifying the metadata itself. Even if **(i)** can be disallowed at the contract level, metadata hosted on third-party (web) domains can be freely modified by $\mathcal{A}$, if she controls the domain. This second attack can be prevented if the metadata is hosted in IPFS. Since the URL of an object stored in IPFS includes the hash of its content, the metadata cannot be modified while retaining the same URL recorded in the NFT.

For internal token contracts, CryptoPunks, Foundation, Rarible, and Nifty offer no way to update the `metadata_url` of an NFT. Axie allows the creator to modify the URL at any time. OpenSea, SuperRare, and Sorare allow modification by the creator until the first sale. Since only Foundation mandates storing the metadata on IPFS, other NFTMs are susceptible to the second attack for the internal contracts. Since no NFTM supporting external token contracts employs any check to prevent metadata tampering, both attacks are feasible.

▶ **Quantitative analysis.** We performed a differential analysis to determine the change in the `metadata_url`s of external assets over a period of time. Specifically, we monitored the `metadata_url`s of all 9,064,767 external OpenSea assets three times over a span of six months in an uniform interval—in June 2021, September 2021, and December 2021, respectively. Since ERC-721 metadata extensions are optional (explained in Section 3.5), `metadata_urls` were completely missing for some of the assets. Also, OPENSEA took down some assets during this time period, which is why their `metadata_urls` could not be retrieved in the subsequent crawl. After excluding these two kinds of assets, we were left with 3,079,139 assets that had `metadata_urls` in all three crawls. According to our observation, the `metadata_urls` of 89,089 (2.89%) and 35,446 (1.15%) assets changed between the first two and the last two crawls, respectively.

3.4.3 Token Listing

(L1) Principle of least privilege. While listing an NFT, the NFTM takes control of the token so that when a sale is executed, it can transfer the ownership of the NFT from the seller to the buyer. To this end, the NFTM needs to be either **(i)** *the owner* of the NFT: that is, the current owner transfers the asset to an escrow account $\mathcal{E}$ during listing, or **(ii)** *a controller*: an Ethereum account $\mathcal{C}$ that can manage that specific NFT on behalf of the owner, or **(iii)** *an operator*: an Ethereum account $\mathcal{O}$ that can manage all the NFTs in that collection. The escrow model in case **(i)** is risky because one single escrow contract/wallet $\mathcal{E}$ managed by the NFTM holds all assets being traded on the platform. Therefore, the security of all assets in a marketplace depends on the security of the escrow contract or the external account that manages such contract. This design essentially violates the principle of least privilege. As a result, either a vulnerability in the contract or a leak of the private key of the external account could compromise the security of all the stored NFTs. NIFTY, FOUNDATION, SUPERRARE follow this approach. A safer alternative would be to adopt **(ii)** or **(iii)**, where a proxy contract $\mathcal{C}$ or $\mathcal{O}$ deployed by the NFTM becomes the controller of the NFT, or the operator of the entire NFT collection, respectively. As enforced by the marketplace contract, the NFTM is able to transfer an NFT only when it has been put on sale and the required amount is first paid to the seller. This ensures the safety of the NFT token even in case of a marketplace hack. If the private key of a seller (owner of an NFT) gets leaked, it can, at most, compromise the safety of that specific NFT or collection, as opposed to all the NFTs as in the case of the escrow model.

▶ **Quantitative analysis.** Among the NFTMs in our dataset, FOUNDATION holds tokens in an escrow contract, while NIFTY uses an Externally Owned Account (EOA) as escrow wallet. SUPERRARE escrows tokens only when an auction is ongoing. The larger the number of NFTs held in escrow, the greater is the risk. On December 31, 2021, SUPERRARE, FOUNDATION and NIFTY held 55, 64079, and 90988 NFTs in their escrow accounts, respectively. In Appendix 3.8, we show how the number of escrowed NFTs increased over time for both NFTMs.

(L2) Invalid caching. While displaying an NFT on sale, OPENSEA and RARIBLE leverage a local caching layer to avoid repeated requests to fetch the associated images. If the image is updated, or disappears, the cache goes out of sync. This could trick a buyer into purchasing an NFT for which the asset is either non-existent or different from what the NFTM displays using its stale cache.

▶ **Quantitative analysis.** To understand the potential impact of this caching issue, we measured how many `image_urls` in our OPENSEA dataset are inaccessible (non-200 `HTTP` response code), but OPENSEA still serves the corresponding cached versions. Out of total 12,215,650 NFTs, `image_urls` of 3,945,231 (32.30%) tokens were inaccessible. However, OPENSEA still cached 2,691,030 (68.21%) of those inaccessible images, thus creating the illusion that the asset linked to the NFT is still alive. One such broken collection is GODS UNCHAINED, a verified collection with an overall trading volume of 19.8K Ethers.

			Count	Total Sales	Average Sales	Taken Down
OpenSea	Seller	Verified	502	$114.5M	$228,028	-
		Non-verified	124,398	$2.7B	$22,001	-
	Collection	Verified	1,805	$3.3B	$1,824,882	88
		Non-verified	234,112	$403.4M	$1,723	11182

Table 3.3: Number of verified and non-verified sellers and collections, along with corresponding sales volumes.

(L3) Seller and collection verification. Listings by verified sellers/collections are not only given preferential treatment by the NFTMs, but they also attract greater attention from the buyer community. However, the verification mechanism is typically ad-hoc, and the final decision is at the discretion of the NFTMs. Common requirements include sharing the social media handles of the sellers and proving their ownership, sharing contact information, collections needing to reach certain trading volume, submitting the draft files of the digital artworks, *etc.* Marketplaces such as FOUNDATION adopt a stricter policy by mandating verification of all the sellers on their platform. However, there are NFTMs, *e.g.*, OPENSEA, RARIBLE, where verification is optional. Buyers are expected to exercise self-judgment when trading on these platforms, which, unfortunately, puts them at greater risk.

Since verification comes with financial benefits, it has been abused in different ways: **(i) Forging verification badge.** Scammers forged profile pictures with an image of the verification badge overlaid on them, making the profiles appear visually indistinguishable from the verified ones at a cursory glance. **(ii) Impersonation.** Abusing weak verification procedures, scammers got their fake profiles verified by just submitting social media handles, without actually proving the ownership of the corresponding accounts [120]. **(iii) Wash trading.** One of the requirements of OPENSEA to verify a collection is to have at least 100 ETH in trading volume [142], which is possibly hard to attain for a newly launched collection. Historically, this requirement has incentivized people to perform *wash trading*, *i.e.*, performing fictitious trades between multiple accounts that are all under the control of the attacker, to artificially inflate sales volumes.

▶ **Quantitative analysis.** To highlight the economic incentive behind verification abuse, we present the number of sales and the sales volume generated by the verified and non-verified sellers and collections in OpenSea in Table 3.3. Though only 0.40% sellers and 0.77% collections of OpenSea are verified, the average sales per verified seller and collection are 10 and 1,059 times more that their non-verified counterparts, respectively.

Next, we measure how effective the NFTM verification mechanisms are in preventing abuse. Had the verification mechanism been foolproof, then a verified collection could not be malicious, and in turn, it should never have been taken down. However, we observed that 4.88% of the verified and 4.78% of the non-verified OpenSea collections were taken down in six months (between June and December 2021). This indicates that though verification attempts to reduce abuse, it fails to eliminate it completely. The fact that the verified collections are still taken down shows that bad actors do "slip through" the system and verify their collections.

3.4.4 Token Trading

(T1) Lack of transparency. NFTs are asset-ownership records that should be stored on the blockchain to allow for public verifiability. In a decentralized setting, an NFT sale is handled by a marketplace contract $\mathcal{C}_m$ that invokes the `transfer()` API of the token contract $\mathcal{C}_t$ to transfer the token from the seller to the buyer. Every *sale* transaction and the associated *transfer*, for example, the `atomicMatch()` call in case of OpenSea, is visible on the blockchain. Among other things, each transaction includes the following information: **(i)** address of the seller (current owner), **(ii)** address of the buyer (new owner), **(iii)** how much the NFT was sold for, **(iv)** time of ownership transfer. Querying for ownership has further been made easier by ERC-721 `ownerOf()` API that returns the current owner of a token. The sales records, in conjunction with the API, permit one to reconstruct the precise sales and ownership history of an NFT.

On the other hand, if sales records and transactions are stored off-chain, it becomes impossible to verify any trades and the ownership history of an NFT. Moreover, a malicious NFTM can abuse this fact to forge spurious sales records to inflate the trading activity and volume. Off-chain records are susceptible to tampering, censorship, and prone to disappear if the NFTM database goes down. Among the NFTMs we surveyed, only Nifty maintains off-chain records. When an item is listed, Nifty takes control of the NFT by first having it transferred (T_1) to an escrow wallet. Thereafter, multiple trades can take place while Nifty holds the custody of the asset, but no sales record is ever emitted on the blockchain. If and when the owner decides to take the NFT out of Nifty, the marketplace transfers (T_2) the token back to the owner's account. Since only T_1 and T_2 are visible from the blockchain, no intermediate ownership and sales activity can be verified.

(T2) Fairness in bidding. NFTMs implement bidding either **(i)** on-chain, through a smart contract that requires the bid amounts to be deposited while placing the bid, or **(ii)** off-chain, through the NFTM dApp which maintains an orderbook without requiring any upfront payment. Off-chain bidding is *unfair* as it can be abused by both the NFTM and the users. Since bids are not visible from the blockchain, NFTMs can inflate the bid volume to create hype. Also, placing bids is inexpensive, as there is no money transfer involved. Therefore, such NFTMs are more susceptible to *bid pollution*, a form of abuse where a large number of *casual* bids are placed on items. Since no money is locked, most of these bids are likely to fail due to a shortage of funds in the bidder's account at the time of execution. Since on-chain bidding costs gas to place/cancel bids, it deters scammers from placing spurious bids, making abuses less frequent. Moreover, on-chain bids reserve the bid amount upfront. Therefore, such bids invariably succeed during settlement. In OpenSea, we observed sellers complain that (attempted) sales of their items fail because the WETH balances of the winning bidders drop below the offered amounts.

▶ **Quantitative analysis.** Unless a bid fails due to lack of funds, an NFT gets transferred to the highest bidder at the end of the auction. To measure the extent of bid pollution in the NFTMs, we enumerated the auctions where the highest bidder did not receive the item. This is unfair to the seller, because the bid immediately below might be a lowball offer. Our analysis uncovered 16,215 and 15,368 such instances out of 48,862 and 19,109 total auctions in OpenSea and Rarible, respectively. We did not find any evidence of the same in the Foundation, Axie, and SuperRare marketplaces.

(T3) Royalty distribution and marketplace fee evasion. If a royalty is set, every trade should earn a fee for the creator. However, we identified ways in which users can potentially abuse the royalty implementations: **(i) Cross-platform.** As explained in Section 3.2, royalty is enforced by either the marketplace contract or the dApp, both of which are specific to an NFTM. Also, NFTMs do not share royalty information with each other. Therefore, royalty set on one platform is not visible from the other. Leveraging this lack of coordination, a malicious seller can evade royalty by trading the NFT through a platform where royalty is not set, though it is set on another. **(ii) Non-enforcement.** Neither royalty nor marketplace fees are enforced in ERC-721 token contracts. A malicious seller can thus avoid both payments by transferring (ERC-721 `transfer()`) the NFT to the buyer directly and settling the payment off-platform. Both royalty and fees could be levied inside the `transfer` method of the token contract, though the additional logic makes the API more expensive. **(iii) Post-sales modification.** OpenSea and Rarible allow the creator to modify the royalty amount even after the primary sale. Now, the royalty is calculated on the price listed by the seller. In a potential abuse scenario, a creator can first lure a buyer B by setting a low royalty and then increasing it post-sales. During secondary sales, B may not notice this change at all, and may end up giving more royalty to the creator than initially advertised.

▶ **Quantitative analysis.** We discovered potential abuses of unconditional token transfer **(case ii)** to evade NFTM fees and royalty. The question of evasion appears when a seller S lists an NFT on a marketplace to gain popularity, but executes the trade off-platform, entirely bypassing the marketplace protocol. There could be two possible cases. Seller S might trust the buyer B and, therefore, transfers the NFT first. After that, B settles the payment. In the other case, the order is reversed. For the assets listed in each NFTM, we counted the number of occurrences on the blockchain where an address (seller) S transferred the NFT to another address (buyer) B, and B sent a payment to S on-chain within 15 minutes (before or after) the transfer transaction. We found 56920, 302, 2777, 5, 814, 56, and 0 such instances for assets listed in OpenSea, SuperRare, Rarible, Foundation, CryptoPunks, Sorare, and Axie, respectively. Note that this estimate is conservative, because the payment could be made either off-chain or outside the time window that we considered for our analysis.

We also measured how often creators abuse sellers by increasing the royalty after the primary sale **(case iii)**. For each OpenSea asset, we enumerated the "sell" events in increasing order of time, and counted the number of times the royalty was increased with respect to the previous sale. We discovered 157,450 instances of such royalty modifications across 20,802 (8.81%) collections.

3.5 Issues Related to External Entities

The asset (picture, video) that an NFT points to must be accessible for this NFT to be "meaningful." NFTs can point to assets in two ways. If the NFT contract is ERC-721-compliant and implements the metadata extension, then the token includes a `metadata_url` on-chain, which points to a metadata record (JSON). This record, in turn, includes an `image_url` field that points to the actual digital asset. Many older tokens, on the other hand, are not standard-compliant and do not contain any on-chain `image_url`. Instead, they use some ad-hoc, off-chain scheme to link to an asset. For such NFTs, NFTMs implement custom support so that they can generate valid image URLs. Since both the metadata record and the asset are stored off-chain, those do not enjoy the same guarantee of immutability as the NFT itself. When any URL becomes inaccessible, that breaks the link between the NFT and the corresponding asset. In practice, the URLs frequently point to a distributed storage service, *e.g.*, IPFS, or centralized storage, *e.g.*, a web-domain or Amazon S3 bucket. For IPFS URLs, if the NFT owner is aware, she can keep the NFT "alive" by pinning the resource (*i.e.*, storing it persistently). Even that could also be problematic, because NFTs do not store the hash value of the actual resource but rather store URLs that point to an IPFS gateway web service. If the gateway becomes unavailable, the NFT "breaks." In general, NFTs that include URLs that point to domains outside the control of the NFT owners risk getting invalidated when the corresponding domains go away.

▶ **Quantitative analysis.** We performed an analysis to quantify the number of OpenSea NFTs that were "lost" due to the reasons outlined above. As of June 15, 2021, out of our 12,215,650 assets from OpenSea, there were only 3,175,644 assets with a valid `metadata_url` field. Querying OpenSea's API, we obtained 8,363,550 assets with non-empty `image_url` fields. The remaining 3,860,607 assets did not have an `image_url` field, which means that they are hosted directly on OpenSea (content creators have the option to leave the image URL field empty, in which case OpenSea handles the hosting). We first check whether the image and metadata URLs point to resources hosted on IPFS. Next, we check whether the URLs are still accessible. To this end, we perform an `HTTP HEAD` query. If the query returns with a response code other than 200 (`OK`), we perform

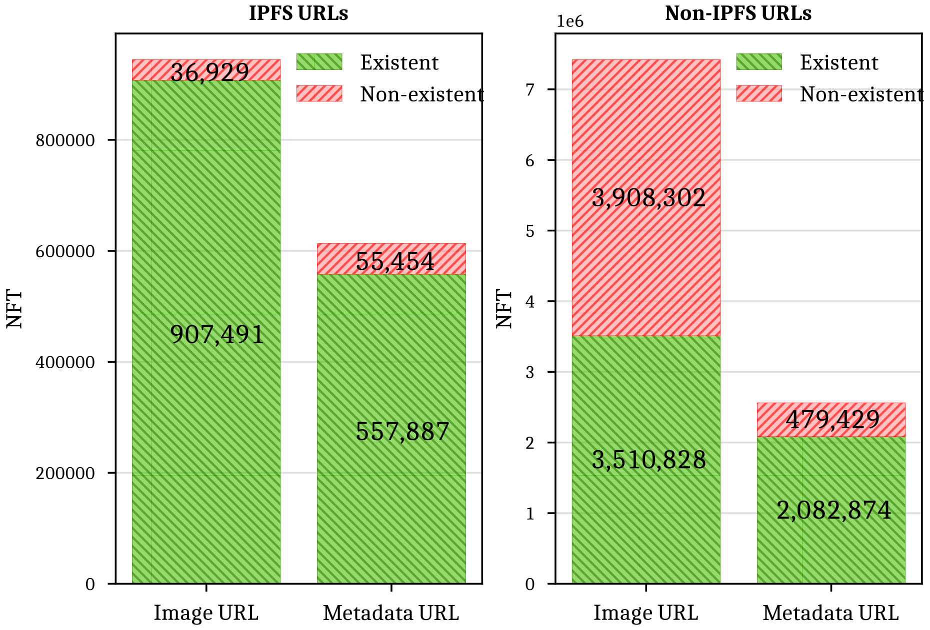


Figure 3.3: Validity of image and metadata URLs

an `HTTP GET` query next. If that also returns a non-200 response code, we mark that URL as *inaccessible*. We take this two-step approach to optimize for performance, and not to generate false negatives due to web servers that do not respect `HEAD` queries. Also, the servers hosting the assets could be offline at the time of testing, but later come back up online. To account for this possibility, we repeated the above URL-check three times in a span of 15 days. Only the assets marked as *inaccessible* in the previous attempt were tested for accessibility each time. An asset is finally marked as *inaccessible* only if all three attempts agree.

Figure 3.3 reports our findings. Two important observations are: **(i)** Only 3.91% of the assets (images) and 9.04% of metadata records hosted on IPFS have disappeared in our dataset between June and December; as expected, NFTs hosted on IPFS are less likely to disappear than those hosted on non-IPFS domains. **(ii)** Though IPFS is supposed to be more resilient to disappearance of the assets, a majority of asset URLs (88.71%) as well as metadata URLs (80.69%) are hosted on non-IPFS domains. Looking at all lost NFTs, they have generated a staggering amount of $160,761,805 USD in revenue from

118,294 transactions. Not only that, due to the caching issue we discussed in Section 3.4, it is very well possible that a fraction of them are still in circulation. As this analysis shows, persistence is a pressing issue in the NFT space.

3.6 Fraudulent User Behaviors

In this section, we study the impact of various fraudulent user activities that occur in NFTMs. In particular, we look at counterfeit NFT creation as well as trading malpractices, such as wash trading, shill bidding, and bid shielding. In Appendix 3.13, we then cover a few more types of malicious activities that were reported in blogs and articles.

3.6.1 Counterfeit NFT Creation

The authenticity of an NFT is endorsed by the smart contract managing the collection. Therefore, to ensure that the token one is buying is legitimate, buyers are advised to verify the contract address of the collection from official sources, *e.g.*, the project's web page, before making a purchase. Unfortunately, buyers are not always aware of the existence of counterfeits, or of how they can verify an NFT's authenticity. Instead, they only rely on the names and visual appearances of items in the marketplaces. This makes it possible for malicious users to offer "fake" NFTs. We observed the following types of counterfeits:
(i) Similar collection names. There are fake NFTs that use the name of a collection or individual piece that resembles the original (victim) one. A common trick is to substitute ASCII characters in the original name with non-ASCII characters that look alike. To prevent such abuse, OpenSea restricts users from using popular collection names and certain special characters. Still, it is often possible to circumvent these limitations, *e.g.*, by adding a dot(.) at the end of the name or substituting an upper-case character with a lower-case one, *e.g.*, a fake of "CryptoSpells" collection used the name "Cryp-

tospells." Moreover, restrictions can cause problems for legitimate users, *e.g.*, French users complained about not being able to use the accented characters in collections.

(ii) Identical image URLs. Some fake NFTs point to existing assets, *i.e.*, they simply copy the `image_url`s of legitimate NFTs. For example, CryptoPunks is a well-known collection. Of course, nothing prevents a scammer from deploying her own token contract on the blockchain and mint tokens that point to CryptoPunks. A buyer who just looks at the appearance of items in a collection will see the CryptoPunks images and might mistake the NFTs for the originals.

(iii) Similar images. Instead of copying the `image_url`, a scammer might copy the digital asset and then mint an NFT that points to this copy. As of now, no NFTM runs any similarity check to detect if a media file has already been used by other NFTs.

▶ **Quantitative analysis.** We looked for each type of counterfeits present in the OpenSea dataset comprising of 12,215,650 NFTs spread across 236,057 collections.

(i) To check for (potential) counterfeits that abuse similar collection names, we compute the *Levenshtein distance*, an *edit distance* metric between pairs of collection names (strings). Since a shorter distance indicates greater similarity, we considered a maximum distance of 2 characters, which means that we only consider collection names as similar if they differ in at most two characters. We considered 52,399 collections that have names longer than 7 characters, and a minimum of 10 NFTs in it (collections with fewer NFTs could be insignificant) to avoid spurious matches. Given that it is more beneficial to imitate verified collections, we only considered collection pairs that include one verified collection (and the other one is considered to be its *replica*).

Our analysis found 322 collection pairs with similar names. We noticed that the names of most of the replica collections were minor modifications of the names of the respective verified collections, for example, pluralizing a noun, adding whitespace at hard-to-notice positions, *etc.*, which indicates a potential intent to mislead. We then randomly picked

100 pairs and checked if those replica collections indeed contain images that are similar to the verified ones and that could mislead buyers. Since judging the similarity visually could be subjective, two researchers independently performed the assessment, and a pair was marked "visually similar" only if both the decisions agreed. We discovered 11 such collections, which we reported to OpenSea requesting a take-down. Moreover, we identified an additional 11 collections that were already taken down by OpenSea (which indicates wrongdoing) between June and December 2021.

(ii) To check for counterfeits that leverage identical `image_url`s, we first collected 8,363,550 `image_url`s comprising of 944,420 IPFS, and 7,419,130 non-IPFS URLs from our dataset. Objects on IPFS are accessed through IPFS gateways, which are web services. An IPFS URL is typically of the form: `http(s)://<gateway>/<ipfs_hash>`. Any `gateway` can be used to access the object pointed to by `<ipfs_hash>`. Therefore, we pre-processed those URLs to extract only the hash component. In the last step, we performed a string comparison between every pair of IPFS hashes and non-IPFS URLs, which reported 356,377 and 2,082,119 identical IPFS, and non-IPFS URLs with at least one duplicate, respectively.

(iii) To find potential counterfeits due to image similarity, we crawled the images pointed by the `image_url` for all NFTs in our dataset. Since the individual assets linked to NFTs can be very large, we decided to focus on downloading just the smaller resolution version of an asset generated and cached by OpenSea. We then used the *perceptual* algorithm [143] of ImageHash [144], a popular (2.1K GitHub stars) image hashing tool, to compute a "fuzzy" hash that is tolerant to small perturbations of the images. Lastly, we compare every pair of hashes to find similar images. We refrain from comparing hashes of the images that are part of the same collection, as they are likely similar (but not counterfeits). We downloaded 9,991,013 images, and we discovered 59,425 hash collision pairs. We randomly picked 100 such pairs, and manually verified that 90% of those image

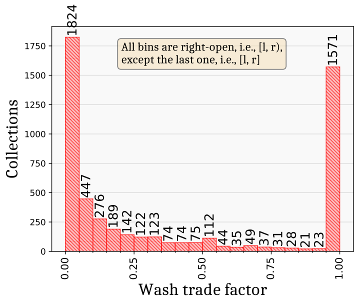


Figure 3.4: Distribution of wash trading factors across collections

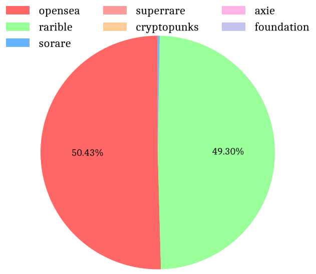


Figure 3.5: Relative volumes of wash trading in different marketplaces

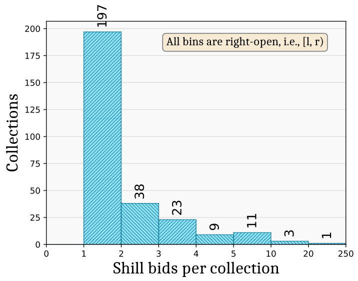


Figure 3.6: Distribution of shill bidding across collections

pairs are indeed visually identical.

3.6.2 Trading Malpractices

In this section, we explore illicit trading practices, specifically, wash trading, shill bidding, and bid shielding [145–147]. We first discuss how these malpractices are relevant in the context of NFTMs, and then build heuristic models to detect such attacks. Finally, we apply these models to all 13,628,411 assets and 354,535,763 events we collected (Section 3.3). The goal is to measure the extent and impact of these trading activities on the top 7 NFTMs.

Data modeling. From the event data and the Ether flows collected from blockchain transactions, we extract actions (such as transfers, sales, bids, ...) that operate on NFTs. Figure 3.7 shows the types of predicates (actions) that we record for users u, assets a, auctions id and prices p. These predicates capture relationships that we use to build four different graphs: A sales graph $\mathcal{G}_s$ (sale), a bidding graph $\mathcal{G}_b$ (auction, bid, cancel_bid, win), a payment graph $\mathcal{G}_p$ (paid), and an asset transfer graph $\mathcal{G}_t$ (transfer). $\mathcal{G}_b$ contains two types of nodes: users (u) and assets (a), and directed edges from u to a annotated with property tuples of the form (p, t, id). All of $\mathcal{G}_s$, $\mathcal{G}_p$, and $\mathcal{G}_t$ contain only one type of node:

$\mathsf{sale}(u_1, a, p, t, u_2)$:− u_1 sold a to u_2 at price p at time t
$\mathsf{auction}(u, p, t, id, a)$:− u started auction with id id at time t with starting price p
$\mathsf{bid}(u, p, t, id, a)$:− u placed bid p on a at time t on an auction with id id
$\mathsf{cancel\_bid}(u, p, t, id, a)$:− u canceled bid p on a at t on an auction with id id
$\mathsf{win}(u, p, t, id, a)$:− u won auction id on a at time t with price p
$\mathsf{paid}(u_1, e, u_2)$:− u_1 transfered e ethers to u_2
$\mathsf{transfer}(u_1, a, u_2)$:− u_1 transfered a to u_2

Figure 3.7: Relationships in graphs $\mathcal{G}_s, \mathcal{G}_b, \mathcal{G}_p, \mathcal{G}_t$.

users (u), and directed edges from u_1 to u_2. Edges in $\mathcal{G}_s$, $\mathcal{G}_p$, and $\mathcal{G}_t$ are annotated with property tuples of the form (a, p, t), (e), and (a), respectively.

Wash Trading

In wash trading, the buyer and the seller collude to artificially inflate the trading volume of an asset by engaging in spurious trading activities. In NFTMs, users wash trade to either create the illusion of demand for a specific asset, artist, *etc.*, or to inflate metrics that are of their financial interest, such as getting a profile/asset verified, or collecting rewards. For example, RARIBLE users are incentivized by \$RARI governance tokens where the more a user spends, the more tokens they receive [148]. It is suspected that many high-value NFT sales related to popular projects such as CRYPTOKITTIES [126] and DECENTRALAND [149] are instances of wash trading [146].

Detection. In the NFT space, wash traders primarily intend to increase the sales volume of NFT collections. To detect wash trading, given a set of assets $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ that are part of a collection, we check for a set of users (addresses) $\mathcal{U} = \{u_1, u_2, ..., u_m\}$ who heavily trade those assets with each other. We assume a limited number of colluding users to make the problem tractable (we use an empirical threshold of 50 users). In other to generate wash trades, these users repeatedly trade that set of assets among them, which often results in cycles in the sales graph $\mathcal{G}_s$. Hence, we check $\mathcal{G}_s$ for the existence of cyclic relationships among these users. In a strongly connected component (SCC) of a graph, there exist paths between all pairs of vertices. Therefore, this type of wash trade can be detected [150] by checking if two users: u_1 and u_2 appear in any SCC of the sales graph $\mathcal{G}_s$. In other words, if $\mathsf{SCC}(u_1, u_2, \mathcal{G}_s)$ holds, it means that both the users are involved in round-trip trades, *i.e.*, there exist either direct, or indirect sale relations between them in both the directions. Now, two users being a part of an SCC can be accidental, and does not indicate the frequency of trades between them. However, in a wash trade, users are involved in frequent sales. Therefore, we only consider SCCs where the number of sale relationships between every two intermediate users is above (indicating 'heavy' trading volume) an empirically determined threshold (ϵ). We use $\epsilon = 10$ in our analysis.

However, bad actors can come up with more intricate strategies to conceal apparent connections so that such simple detection can be evaded. We manually analyzed the blockchain transactions history and found two evasion strategies that would throw off the prior analysis. In the first case, when we investigated an otherwise legitimate-looking sale relation $u_i \rightarrow u_j \rightarrow u_k$, we realized that both u_j and u_k are funded (Ether transfer) by the same "parent" user u_i. We capture this case by checking if two users: u_1 and u_2 appear in any weakly connected component (WCC) of the payment graph $\mathcal{G}_p$. In other words, if $\mathsf{WCC}(u_1, u_2, \mathcal{G}_p)$ holds, it means that direct or indirect Ether-flow exists between those two users in either direction. In the second case, for a sale relation $u_i \rightarrow u_j \rightarrow u_k$, we identified multiple unconditional asset transfers (ERC-721 `transfer()`) from u_i to u_k, giving a strong indication of a close tie between those users. We capture this case by checking if two users: u_1 and u_2 appear in any WCC of the transfer graph $\mathcal{G}_t$. In other words, if $\mathsf{WCC}(u_1, u_2, \mathcal{G}_t)$ holds, it means that direct or indirect unconditional asset transfer relationships exist between those two users in either direction.

To summarize, our model considers any $\mathsf{sale}(u_1, \_, \_, \_, u_2)$ relation a potential wash trade if: $\mathsf{SCC}(u_1, u_2, \mathcal{G}_s) \lor \mathsf{WCC}(u_1, u_2, \mathcal{G}_t) \lor \mathsf{WCC}(u_1, u_2, \mathcal{G}_p)$.

▶ **Quantitative analysis.** We detected 9,393 instances of wash trading that generated \$96,858,093 USD in trading volume across 5,297 collections involving 17,821 users in all NFTMs except AXIE, FOUNDATION, and CRYPTOPUNKS. Moreover, out of 238,180 collections in our dataset, only 8,869 collections had more than \$2K in trading volume, out of which 2,569 (28.97%) collections show signs of wash trading.

We define *wash_trade_factor* (WTF) as the fraction of the total trading volume of a collection generated by wash trading, *i.e.*, if WTF is 1, then all the trades are wash trades. In Figure 3.4, we show the distribution of the *wash_trade_factor* across collections where wash trading has been detected. Of all the wash traded collections, 1,824 (34.43%) collections had less than 5% (WTF < 0.05) of the trades generated by wash trades. Interestingly, we discovered 1,571 (29.66%) collections which were heavily abused, because more than 95% of all of their trades are wash trades, totaling \$3,407,284 USD in the trading volume. Figure 3.5 shows the relative volumes of wash trades that have happened in different NFTMs. Though nearly equal volume of wash trades were discovered in both RARIBLE (49.30%) and OPENSEA (50.43%), given that the overall trading volume of OPENSEA is 21 times more (Table 3.1) than that of RARIBLE, it seems that wash trading is significantly more frequent in RARIBLE than OPENSEA. Our finding is also corroborated by discussions we saw on RARIBLE DISCORD, which indicates a heavy amount of past wash trading incidents as malicious users attempted to secure \$RARI tokens.

▶ **Manual analysis.** In our analysis, the size of a connected component represents the number of addresses involved in a wash trade. We observed that 98.88% (9,288 of 9,393) of the reports had a component size at most 10. Therefore, for our manual analysis, we randomly selected 100 reports, and checked whether one of the following two conditions holds: **(i)** if a addresses are involved in t transactions on n NFTs, then both $t \geq 2a$ and $n \ll t$ need to hold. The intuition is that if a set of users "heavily" trades on only a small number of assets, then those are likely to be wash trades. Alternatively, **(ii)** the addresses involved in trading are all funded by a common, on-chain funding source. This is true when the "supposed" buyers, in reality, are all funded directly by the seller, or by a seller-controlled address, before making (pseudo) purchases. If one of these two conditions holds, we consider a detected wash trade instance as a true positive. We determined all the sampled instances as true positives.

Limitation. Ethereum mixers (informally "tumblers"), such as Bitmix [151], ETH Mixer [152], and Tornado Cash [153], are anonymity services that help to conceal the true source of a payment by breaking the link between the receivers and the sender of the funds. Specifically, these services accept Ethers from a user, and either route it to a smart contract, or relay it through a complex, large network of addresses by splitting the amount into a number of micro-transactions; essentially mingling that fund with hundreds of other users. Since our wash trade detection strategy leverages information about Ether flows between two addresses, mixers can lead to false negatives.

Shill Bidding

Shill bidding is a common auction fraud where a seller artificially inflates the final price of an asset either by placing bids on her own asset, or colluding with other bidders for placing spurious bids with increasingly higher bid amounts. This can lead to honest bidders paying higher prices than they would have otherwise. With high-value bids on assets becoming increasingly common, it is suspected that many sales suffer from artificial price inflation [147].

Detection. Detecting shill bidding is difficult when looking at a single auction in isolation. It becomes even harder when malicious users take turns, placing bids on each others' auctions so that the seller-bidder relation changes. In this work, we only consider the simple case where a specific user repeatedly places bids in auctions, yet never (or rarely) purchases anything. Moreover, we check whether there is some relationship between this user and the seller. Thus, our findings should be viewed as a lower bound on the actual number of shill bidding occurrences in NFTMs. Our detection mechanism draws on our insight from the manual analysis of NFTM activities and prior work [154].

Let $\mathsf{bid}(u_b, p_i, t_i, id, a)$ denote the i-th bid placed by user u_b with amount p_i at time t_i on asset a in an auction with id id created by the seller u_s. Then, user u_b is a shill bidder if:

Rule 1. u_b places at least n bids on an asset a auctioned by u_s with monotonically increasing bid amounts. That is, $\forall i \in [1, n], \mathsf{bid}(u_b, p_i, t_i, id, a)$, the following holds: $\forall i, \forall j, \;\; t_i > t_j \implies p_i > p_j$

Rule 2. u_b never buys the asset a, *i.e.*, $\mathsf{win}(u_b, \_, \_, id, a)$ is false .

Rule 3. u_b has limited buying/selling activity, *i.e.*, $|\{\mathsf{sale}(u_b, \_, \_, \_, \_)\} \cup \{\mathsf{sale}(\_, \_, \_, \_, u_b)\}| < \sigma$, where σ is an empirically determined threshold. We set $\sigma = 10$ for our analysis.

Rule 4. u_b is "connected" to the seller u_s either through Etherflows ($\mathcal{G}_p$) or asset transfers ($\mathcal{G}_t$). That is, $\mathsf{WCC}(u_b, u_s, \mathcal{G}_t) \vee \mathsf{WCC}(u_b, u_s, \mathcal{G}_p))$ holds.

Rule 5. We define *shill score* as the ratio of the number of times u_b participates in an auction created by u_s and the total number of auctions that u_b participated in. In our detection approach, the *shill score* must be greater than μ, another empirically determined threshold. We set $\mu = 0.8$ for our analysis.

▶ **Quantitative analysis.** We detected 703 instances of shill bidding across 282 collections involving 1,211 users in all NFTMs except Axie and CryptoPunks. We estimate *shill_profit* as the profit made by the seller due to shill bidding. Specifically, assume legitimate bidders place bids on an item first, and then shill bidding drives the price up. If b_l is the offer made by the last legitimate bidder before shill bidding starts, and the item is finally sold at b_s due to artificial inflation, we compute $(b_s - b_l)$ as the *shill_profit*. According to our analysis, malicious sellers have collected a cumulative profit of $13,014,662 USD from all the shill bidding instances detected. In Figure 3.6, we show the frequency of shill bidding instances discovered across collections where shill bidding has been detected. The majority (197) of the collections have just one instance of shill bidding, while almost all collections (281) have fewer than 20 shill bids.

The one exception is the official collection of FOUNDATION, which seems to be heavily affected by shill bidding. With 212 instances (30.16% of all instances detected) of shill bids in that collection alone, it becomes the one with the most number of shill bids on any individual collection. Our model also reports frequent shill bidding activity on the official collection of SUPERRARE (15 instances) and CRYPTOVOXELS (11 instances), which is an OPENSEA verified collection with 5.8K items and a cumulative trading volume of 19.2K ETH.

▶ **Manual analysis.** Since shill bidding often closely resembles legitimate bidding behavior, it is harder to detect than other malpractices. Therefore, to remain conservative during ground-truth determination, we looked for the following conditions: **(i)** the shill bidder S placed at least 3 bids in an auction, and **(ii)** if the average price of the items that S bought is p, and the average bid that S placed in that auction is b, then $p \ll b$, and **(iii)** S never bought any NFT from that seller. We manually verified 100 reports that we randomly selected from the instances that our approach detected. Out of these 100 cases, 61 show strong indications of being instances of shill bidding. For the remaining 39, we could not draw any definitive conclusion from the trading patterns alone. We observed an interesting shill bidding case in FOUNDATION, where the initial reserve price of an NFT was 2 ETH. The item was targeted by a shill bidder who bid $[3.3, 4.4, 5.5, 6.71, 8.14]$

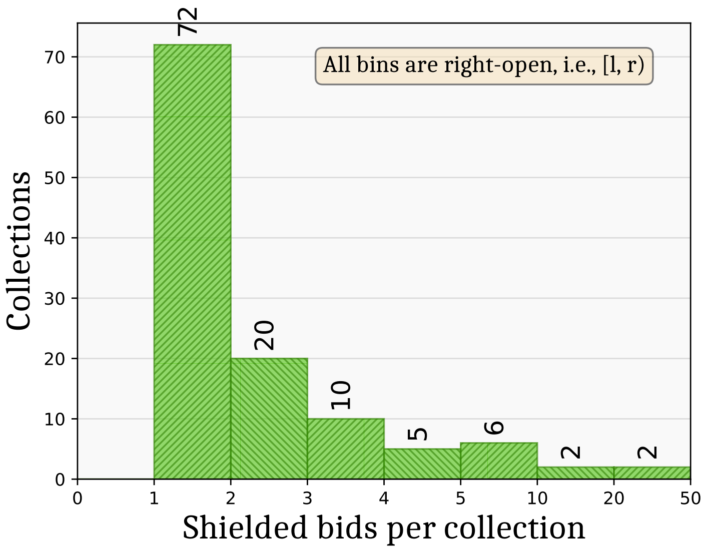


Figure 3.8: Distribution of bid shielding across collections

ETH on that item, thereby making the item finally sell at 9 ETH. However, all the NFTs owned by the bidder were worth between $(0, 2]$ ETH, and the bidder never bought any items from that seller.

Bid Shielding

In bid shielding, a malicious bidder u_2 guards a low bid, possibly from a colluding bidder u_1, with a bid high enough to deter legitimate bidders from placing any additional bids. Immediately before the auction ends, u_2 retracts the bid, thus uncovering the low bid from u_1 to let her win the auction.

Detection. We apply the following heuristics to detect instances of bid shielding in NFTMs. If for two users u_1 and u_2, $\mathsf{bid}(u_1, p_1, t_1, id, a)$, $\mathsf{bid}(u_2, p_2, t_2, id, a)$ and $\mathsf{cancel\_bid}(u_2, p_2, t_3, id, a)$ hold, then u_2 is shielding a bid from u_1 if:

Rule 1. For all bids $\{\mathsf{bid}(u_i, \_, t_i, id, a)\}_{i=1}^{n}$ placed on asset a, t_3 ¿ t_i holds, *i.e.*, no new bid was placed after u_2 retracted her bid on asset a.

Rule 2. u_1 won the auction with id id. That is, $\mathsf{win}(u_1, p_1, t_4, id, a)$ holds, and $u_1 \neq$

$u_2 \wedge p_1 < p_2$.

▶ **Quantitative analysis.** We detected a total 316 instances of bid shielding across 117 collections involving 471 users only in OpenSea. It is expected, because other NFTMs implement bidding policies (Section 3.2) to deter such malpractices, for example, on-chain bids, removal of the the previous bid when outbid, *etc.* We compute *shielded_bid_difference*, the difference in the bid amounts of the two colluding parties, the potential bid shielder and the auction winner. While the minimum *shielded_bid_difference* amount was $200.77 USD, the maximum was as high as $152,606.31 USD for one of the token in Mirandus Vaults, a verified collection. Additionally, all 316 instances together shielded a total of $942,061 USD worth of bids. Figure 3.8 shows the number of instances of bid shielding discovered across collections where bid shielding has been detected. For most of the collections (113 out of 117), we find less than ten instances of bid shielding per collection. Ethereum Name Service (ENS), a popular Ethereum name lookup service, makes it to the top of the list with 49 bid shielding instances. Another notable finding in this category was the CryptoVoxels collection. We noticed several complaints by CryptoVoxels collectors on their Discord server about the recent increase of bid shielding activity. According to our analysis, $24,519.27 USD worth of bids were shielded by 35 instances of bid shielding, which corroborates this prior observation. Our results show that bid shielding is frequent in verified collections as 66.67% (78 out of 117) of the bid shielded collections were verified.

▶ **Manual analysis.** We have manually verified randomly chosen 100 instances flagged by our analysis. During manual analysis, we mark an instance as a *true positive* if **(i)** the potential bid shielder B and the (colluding) auction winner W are the last two highest bidders on that auction in that order, and **(ii)** B cancels her bid just before ($\leq$ 2h) the auction ends, and **(iii)** during the auction, they never outbid each other. Out of the

100 instances, our manual analysis confirms 90 such instances as true positives. Our detection model produced some false positives because it does not take into account the last condition listed above. Let b_i and w_i be the bids from B and W, respectively. Now, first they outbid each other, *i.e.*, $b_1 \rightarrow w_1 \rightarrow b_2 \rightarrow w_2 \rightarrow b_3$, and then B removes b_3 at the last moment (possibly B just changes her mind). This is *not* a bid shielding scenario, as the bids from B drove up the price for W. This would not happen in a bid shielding scenario as B and W are colluding. However, the first two conditions are still met, and therefore our model incorrectly flags this case. We also observed that most of bid shielding activities are performed in verified collections, such as CryptoVoxels, ENS, *etc.*, as they are popular and in high demand.

3.7 Related Work

To the best of our knowledge, we are the first to perform an in-depth study of security and privacy risks in the NFT ecosystem. Our research fits into the recent line of work on cryptoeconomic attacks in decentralized finance (DeFi) systems. The transparency of blockchains opens up the possibility of launching economic attacks by manipulating the market. Since uncommitted Ethereum transactions and their gas bids are visible to other network participants, an attacker can offer a higher gas price to get their malicious transactions mined early in a block, before the victim transaction. This behavior is called *front-running* [8]. The authors in FlashBoys [9] demonstrated how arbitrage bots front-run transactions in decentralized exchanges (DEX) to generate non-trivial revenues. *Sandwich attacks* take this idea a step further by both front- and back-running victim transactions. Zhou *et. al.* [10] quantified the probability of being able to perform such an attack and the profits it can yield. In fact, a recent paper [11] reported the profit extracted from the blockchain to be a staggering \$28.8M USD in just two years, leveraging sandwiching, liquidation, and arbitrage. The authors also measured the prevalence of other profit-making operations, *e.g.*, *clogging* and *private mining*. Another DeFi trading instrument, *flashloans*, allows a borrower immediate access to a large amount of funds without offering any collateral, under the condition that the loan needs to be repaid in the same transaction. Qin *et. al.* [12] analyzed how flashloans have been used to execute arbitrage and oracle manipulation attacks, and they presented a constrained optimization framework to cleverly choose the attack parameters that maximize the profit. DefiPoser [13] proposes trading algorithms to generate profit by crafting complex DeFi transactions, both with and without flashloans. Recent research [14–16] has also characterized and quantified *pump-and-dump*, a price manipulation scheme that attempts to inflate the price of a crypto asset by spreading rumors and misinformation.

3.8 Charts

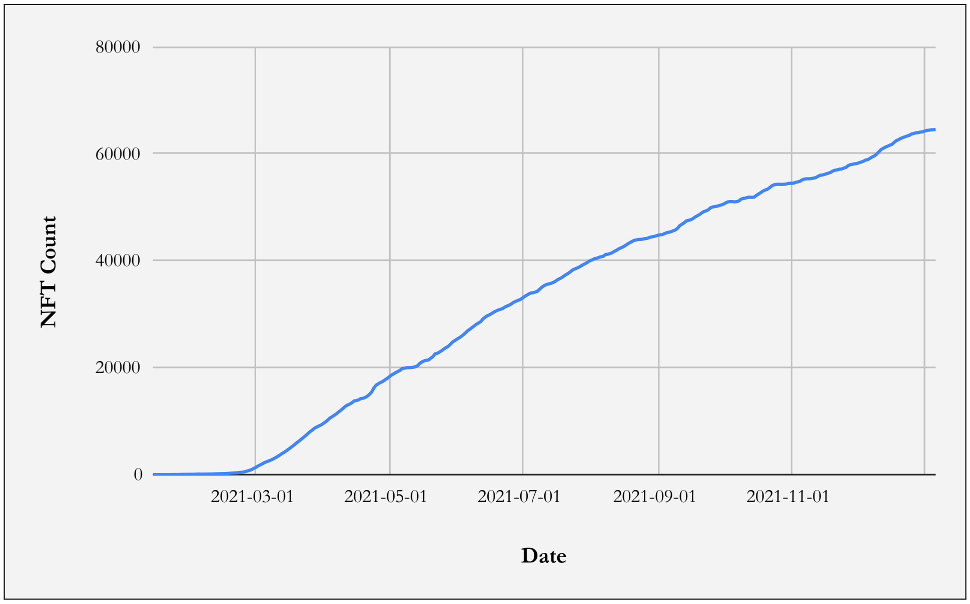


Figure 3.9: Count of NFTs escrowed by Foundation over time

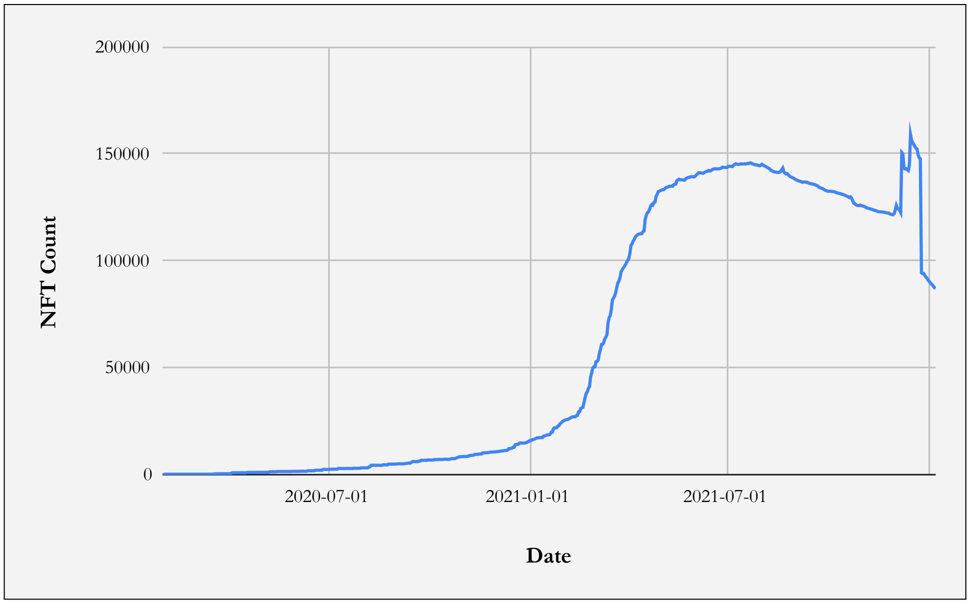


Figure 3.10: Count of NFTs escrowed by NIFTY over time

3.9 Data Collection

In Table 3.4, we provide the details of the types of data, *i.e.*, assets and events we collected from different marketplaces and blockchain.

3.10 Analysis of Top-15 NFT Sales

In this section, we first discuss a few desirable properties of an NFT ecosystem. Then, we analyze the top 15 NFT sales by price with respect to those desirable properties. We report a number of interesting observations with regards to the corresponding sales transactions. Note that collecting information about our high-profile NFT sales was challenging since as is no source of ground truth. We primarily utilized four main sources: **(a)** search engines, **(b)** hashtag search in TWITTER, **(c)** searches on REDDIT, and **(d)** the blockchain.

Entity	Attribute	Description
Asset	Token contract address	Ethereum address of the token contract that manages the asset
	Token ID	Integer ID that uniquely identifies the token among all the tokens managed by the token contract
	Collection name	Name of the collection that the NFT belongs to
	Image URL	URL of the resource (picture/video) that is pointed to by the NFT
	Metadata URL	URL of the metadata JSON if the token is ERC-721-compliant and implements the metadata extension
	Asset listing URL	URL of the listing page of that asset on the NFTM dApp
	Source code availability of the token contract	Boolean flag indicating if the source code of the token contract is available in ETHERSCAN
	Verification status of the collection	Boolean flag indicating if the collection which this asset belong to is verified by the NFTM
Event	Mint (Asset creation)	Minter's (Creator's) address, minting time, asset being minted
	Sell	Seller's address, Buyer's address, Timestamp, Transaction hash, Asset being sold, Sell price (USD, ETH)
	Asset transfer	From address, To address, Asset being transferred, Timestamp, Transaction hash
	Auction start	Asset on which the auction started, Auction creator, Timestamp, Transaction hash
	Bid	Bidder, Asset on which bid is placed, Auction creator, Bid amount (USD, ETH), Timestamp, Transaction hash
	Bid cancel	Bidder, Asset on which bid is canceled, Auction creator, Cancel price, Timestamp, Transaction hash
	Win	Winner, Asset being won, Auction creator, Sell price (USD, ETH), Timestamp, Transaction hash
	Auction end	Asset for which the auction ended, Auction creator, Timestamp, Transaction hash
	ETH transfer	From address, To address, Amount (ETH), Timestamp, Transaction hash

Table 3.4: Details of the data collected for this study.

NFT	**Sale Price**	**Sold on**	**Transferred on**	**Proof of Purchase**
1. Beeple's Everydays	69.30M	03/11/21	03/13/21	✗
2. CryptoPunk #7523	11.80M	06/10/21	07/15/21	✗
3. CryptoPunk #7804	7.56M	03/11/21	03/11/21	✓
4. CryptoPunk #3100	7.51M	03/11/21	03/11/21	✓
5. Beeple's Crossroad	6.66M	02/24/21	N/A	✗
6. Beeple's OceanFront	6.00M	03/23/21	N/A	✗
7. CryptoPunk #5217	5.44M	07/30/21	07/30/21	✓
8. WWW source code	5.43M	–	09/10/21	✗
9. CryptoPunk #7252	5.30M	08/24/21	08/24/21	✓
10. Snowden's StayFree	5.27M	04/16/21	04/16/21	✓
11. Save 1000s of Lives	5.10M	05/08/21	05/08/21	✓
12. CryptoPunk #2338	4.40M	08/06/21	08/06/21	✓
13. Micah's Replicator	4.10M	–	04/27/21	✗
14. Fidenza #313	3.30M	08/23/21	08/23/21	✓
15. Jack Dorsey's Tweet	2.90M	03/22/21	N/A	✗

Table 3.5: Top 15 most expensive NFT sales in descending order of sales price. Dates are in MM/DD/YY format. Missing information is denoted by '–', and N/A denotes that the corresponding event has not taken place yet.

3.10.1 Desirable properties of the ecosystem.

Since NFTs are built around cryptocurrency and blockchain, it is not unfair to expect the ecosystem to draw on the benefits offered by those technologies. We identify the following properties that an NFTM protocol must hold in order to set it apart from traditional e-commerce platforms like Amazon, eBay, *etc.* In other words, an NFTM that lacks in one or more of the following benefits should be deemed less valuable as a new platform—**(P1) Decentralization.** NFTs should be stored on blockchain to ensure persistence, censorship-resistance, immutability, and public verifiability of the asset-ownership record. Keeping verifiability in mind, ERC-721 standard offers the `ownerOf()` API which returns the address of the current owner given a `_tokenId`. Since transfer events alter the ownership of an NFT, all those events should be recorded on-chain for an NFT to be verifiable. Blockchain transaction history allows one to track when the NFT was created, who the previous owners were, and how much it was traded for each time. If an NFTM protocol escrows a token to a wallet W, no sale record is emitted on the blockchain, except the one where the current owner withdraws the token from W to her own wallet. Such a model makes the token opaque and unverifiable. Also, as opposed to blockchain, if the ownership record is stored in a centralized database, it is susceptible to tampering, censorship, and prone to disappear if the database goes away. An NFTM protocol should not sacrifice any of these guarantees in its design. **(P2) Crypto payment.** The payment toward the NFT trade must be made in cryptocurrency, *e.g.*, a primary token like Ether (ETH), or a secondary token like Wrapped Ether (WETH), *etc.* **(P3) Trustless trading.** The trade must happen in a trustless manner, without relying on a third-party T (other than the buyer B and the seller S) who mediates either the transfer of assets or the payment. For example, a protocol where T escrows the token from S, accepts the payment from B, and then exchanges the token and the payment—beats the very purpose of decentralization by making the parties put trust on T. **(P4) Atomic swap.** The transfer of assets and the Crypto payment must take place in the same transaction in an *atomic* manner, *i.e.*, either both succeed, or both fail. Atomicity enables two mutually distrusting parties to get involved in a trade without risking losing the asset or the funds. **(P5) Token minting.** A token has to be minted before or at the time of sales (*e.g.*, lazy-minting), but not after. This must hold because, in a trustless setting, an NFT cannot technically be sold, or transferred, unless it exists when the trade is executed.

3.10.2 Analysis of top sales

In Table 3.5, we consolidate the sales information available in the public domain. As can be seen, at least half of the sales in Table 3.5 violate one or more of our desirable properties. We indicate the principle(s) a specific sale violates inside the parentheses.

For Beeples's Everydays, the highest valued ($69.3M) sale, we found the ownership transfer record one the blockchain. However, payment is not present (**P4**). In other words, we failed to find any evidence that \$69.3M was indeed transferred from MetaKovan (the buyer) to Beeple (the artist and seller).

For the second-largest transaction, several tweets [155, 156] indicated that CryptoPunk #7523 was sold on June 10, 2021. However, the actual transfer on the blockchain took place more than a month later (July 15, 2021) (**P4**). Moreover, the transfer transaction had a value of zero ETH, thus making it impossible to confirm the reported sales amount (\$11.8M).

Beeple's CrossRoad, which was reportedly sold for \$6.6M, was traded on NIFTY, which uses an escrow contract (**P3**). Thus, it does not have any sale or transfer record on the blockchain (**P1**). Likewise, no public history exists for the sale of Beeple's OceanFront (**P1**, **P3**). In fact, querying the blockchain with `ownerOf(_tokenId)` returns the address of the NIFTY gateway as its current owner.

For both the WWW source code and Micah's Replicator, the advertised payment amounts are not verifiable from the blockchain transfer records (**P4**). And TWITTER CEO Jack Dorsey's first tweet was sold on a platform called VALUABLES. This marketplace also uses an escrow contract, and, therefore, no sales or transfer details are publicly available (**P1**, **P3**). At the time of writing, querying the VALUABLES token contract on the Polygon sidechain returns an address owned by the platform as the owner.

3.11 Non-Technical Aspects of the Ecosystem

In this section, we discuss a few non-technical questions surrounding the NFT ecosystem. While we don't claim to be legal experts nor do we offer any tax advise, we frequently encountered certain issues during our work and wanted to bring them to our readers' attention.

Misconceptions around NFT purchase. Here we clarify some frequent misperceptions of the users interacting with the NFT space. **(i) Originality.** Since digital artworks are infinitely and identically reproducible, the "originality" of a piece of art in the NFT world is ascertained by the smart contract managing the corresponding token. Unfortunately, fraudsters have been able to trick victims into buying NFTs pointing to someone else's art, for example, by deploying their own smart contracts. The NFT infrastructure is unable to provide any technical solution to this issue. **(ii) Ownership.** NFTs are used to introduce the concept of "ownership" to digital art. However, what that form of ownership actually means is somewhat subjective to individual's interpretations. When an NFT is purchased, what the buyer really purchases is the NFT "token" on a blockchain. Whether and how that purchase translates to the ownership of the linked digital asset is debatable. **(iii) Copyright.** "Copyright" grants the owner of the copyright the rights to control (a) the manufacture of copies of the original piece, (b) the sale, licensing, or transferring of the copyright itself, and (c) who can produce "derivatives." Merely purchasing an NFT does not transfer the copyright to the buyer. In one common scenario, the buyer posts the linked artwork to social media. This essentially creates a digital copy of the art. Therefore, this might infringe on the copyright of the artist, unless the terms of sale (ToS) explicitly allows the NFT buyer to do so. **(iv) Terms of sale.** A frequent misconception in the NFT space is that the ToS are encoded in the smart contract. Smart contracts are executable code. Therefore, they can enforce certain aspects related to a trade, such as the sales price and royalty. But what if the seller were to include a term that precludes buyers from using the underlying digital art for commercial purposes? A smart contract cannot enforce that provision. A seller would have to resort to traditional methods of enforcement, *e.g.*, demand letters, litigation, *etc.*

Valuation of NFT collections. The value people place in a work of art is largely subjective. In the past, the price of an artwork has typically been decided by community consensus. Moreover, a (valuable) artwork typically has a rich history associated with it. The fact that the NFT market is so young means that such history is not available. As a result, a certain amount of hype (and maybe market maniulation) drives up prices. For example, the descriptions of a large number of NFT projects are rife with hyperbole, and it is not uncommon to find token owners on TWITTER and REDDIT providing long explanations of why a token they own is particularly meaningful. Since NFTs lack any intrinsic value, it is non-trivial for a newcomer to judge its true merit. Hence, they can easily fall prey of such promotions, and sometimes run into exit scams.

Tax implications. Law practitioners seem to agree that the purchase or sale of NFTs is a taxable transfer of property, and is therefore subject to capital gains tax. We noticed users gifting NFTs to others, which might trigger a gift tax. Unfortunately, taxation on NFTs is still a gray zone, as the tax laws are unclear and classic securities laws need to be reapplied. NFTMs are open markets, and cross-border sales can make matters complicated, as NFT buyers and sellers have to deal with different jurisdictions' tax regimes. Also, NFT trades could violate U.S. sanctions law, which prevents U.S. residents or citizens from conducting business with individuals or entities from sanctioned nations. Experts are not even ruling out the possibility of NFTs being used for money laundering to support illicit activities. While taxation regulations are already complicated, none of our examined marketplaces help user to remain tax-compliant by generating tax forms, *e.g.*, 1099K. Instead, we see disclaimers such as OPENSEA's terms of service (ToS), which states: "*You are solely responsible for determining what, if any, taxes apply to your Crypto Assets transactions. Neither OpenSea nor any other OpenSea Party is responsible for determining the taxes that apply to Crypto Assets transactions.*" Given the lack of clarity and support, it is possible for unsuspecting, law-abiding users to inadvertently violate tax rules while interacting with these marketplaces.

Lack of support for selling physical assets. Though NFTs are being used to trade physical assets in limited cases, the current state of the affairs not only violates the basic principles of blockchain sales, but also it gives rise to the potential of an abuse. NFTMs enable two mutually distrusting parties to execute trades in a trustless environment while retaining their anonymity. However, delivery of physical goods requires sharing the details of the buyer with the seller. In a centralized marketplace, *e.g.*, Amazon, the platform itself acts as the trusted third-party (TTP) that protects the buyer information from the seller, often handling the delivery on the seller's behalf. Unfortunately, no current NFTM offers such a service. In fact, even if they would do so, that would violate the spirit of a trustless, peer-to-peer marketplace. The alternative, which is the current practice, is to have the buyer share her contact details directly with the seller. For example, it is not uncommon to find NFTs sold by photographers where they promise the buyer a physical print of the photo, and, therefore, they request the buyer to email their address to the seller. Needless to say, this poses a significant threat to the buyer's privacy. In addition, the absence of a TTP makes arbitration harder in case of any disputes, *e.g.*, non-delivery of the purchased asset. To summarize, no current marketplace protocol satisfies all three desirable yet mutually conflicting requirements, *viz.*, trustlessness, decentralization, and anonymity. Thus, NFT markets are not entirely suitable as platforms to sell physical assets.

3.12 Contract Addresses

We provide the Ethereum addresses of the important contracts used in this chapter in Table 3.6.

Marketplace	Purpose	Contract Address
OpenSea	Marketplace	0x7be8076f4ea4a4ad08075c2508e481d6c946d12b
OpenSea	Token	0x495f947276749ce646f68ac8c248420045cb7b5e
Axie	Marketplace	0xf4985070ce32b6b1994329df787d1acc9a2dd9e2
Axie	Token	0xf5b0a3efb8e8e4c201e2a935f110eaaf3ffecb8d
CryptoPunks	Marketplace	0xb47e3cd837ddf8e4c57f05d70ab865de6e193bbb
CryptoPunks	Token	0xb47e3cd837ddf8e4c57f05d70ab865de6e193bbb
Rarible	Marketplace	0x9757f2d2b135150bbeb65308d4a91804107cd8d6
Rarible	Token	0x60f80121c31a0d46b5279700f9df786054aa5ee5 0xd07dc4262bcdbf85190c01c996b4c06a461d2430 0x6a5ff3ceecae9ceb96e6ac6c76b82af8b39f0eb3
SuperRare	Marketplace	0x2947f98c42597966a0ec25e92843c09ac17fbaa7 0x8c9f364bf7a56ed058fc63ef81c6cf09c833e656 0x65b49f7aee40347f5a90b714be4ef086f3fe5e2c
SuperRare	Token	0xb932a70a57673d89f4acffbe830e8ed7f75fb9e0
Sorare	Marketplace	0xaeb960ed44c8a4ce848c50ef451f472a503456b2
Sorare	Token	0x629a673a8242c2ac4b7b8c5d8735fbeac21a6205 0x9844956f1d45996aa8d322f3483cc58abe34d449 0xd2c98d651a02e34c279ed470a1447a36aa0423ee
Foundation	Marketplace	0xcda72070e455bb31c7690a170224ce43623d0b6f
Foundation	Token	0x3b3ee1931dc30c1957379fac9aba94d1c48a5405
Nifty	Marketplace	off-chain
-	CelebrityBreeder	0xa33ab4b0c9905ebc4e0df5eb2f915bee728b8253
-	Mirandus Vaults	0x495f947276749ce646f68ac8c248420045cb7b5e
-	Gods Unchained	0x0e3a2a1f2146d86a604adc220b4967a898d7fe07

Table 3.6: Ethereum addresses of important contracts.

3.13 Fraudulent User Behaviors - Extended

Digital scarcity. Digital scarcity [157] is the limitation, typically imposed through software, to control the abundance of a digital resource. The more abundant an asset is, the lesser becomes its intrinsic value. Since NFTs are created by smart contracts, it is possible to impose appropriate limitations to ensure scarcity, provided: **(i)** the rarity parameter is stored on-chain, and **(ii)** the contract uses the parameter to prohibit minting beyond promised limits.

Currently, most of the items that claim to be a *limited edition*, or *rare*—it is word of mouth, than any contract-level guarantee. In fact, we found users complaining about a ‘supposed’ limited edition item being minted beyond the promised limit. CryptoMotors, a verified collection in OpenSea, claims to have only 150 GEN1 cars in circulation.

However, the rarity parameter (GEN) is stored off-chain inside the JSON metadata, making it impossible to enforce rarity at the contract level. Additionally, the `totalSupply` parameter, which controls the total supply of a token, is also not fixed, which makes it possible to mint unlimited cars.

Giveaway scams. NFT giveaways are campaigns to distribute free NFTs in exchange for having users promote the newly launched collection on social media. In giveaways scams, scammers lure the users of free NFTs, but ask for 'small' fees to cover the gas cost. In reality, the fee they ask for is several times greater than the gas cost required for the transfer. Sometimes, NFT platforms use a fungible token as the native currency for their services. For example, NFT-ART.FINANCE [158] is powered by their platform token called $NFTART. In some scams, the scammers pretend to put either the NFT, or the platform token 'on-sale'. Users who fall for this send funds to the designated accounts, but never receive the NFT or the tokens in return. Interestingly, there have also been instances where legitimate giveaways were targeted by scammers where they impersonated the 'winner' by faking social media accounts, and had the reward transferred to their wallet, thus forfeiting the real winner.

Front-running. In a front-running attack, an attacker gets a malicious transaction mined before a victim by paying a higher gas price. When a transaction is broadcast in the Ethereum network, it appears in the *mempool*. A replay attack synthesizes a malicious transaction from a profit-making mempool transaction, oftentimes just by copying the arguments verbatim—only to front-run the victim to bag the profit. In reality, automated bots sniff the mempool for such profitable victims. Since NFTs are managed by smart contracts, those are susceptible to front-running.

Also, it has been shown that by merely front-running the `giveBirth` call [159] of the CRYPTOKITTIES token, an attacker would make a profit of $111K USD. In February 2021, an attacker exploited a weakness in CryptoPunks's bid acceptance mechanism [160], for which a bid that was supposed to be closed for 26.25 ETH, returned only 1 Wei ($= 10^{-18}$ ETH) in profit due to being front-run.

Insider trading. An *insider* is one who has access to some confidential information about publicly traded security. Any trade involving an insider is an insider trade. However, it is illegal when the investor leverages that information in deciding when to buy or sell the security, because it gives them an unfair advantage to make a profit from that information. The regulatory gap in the NFT ecosystem surfaced out recently once again when an OpenSea employee was found to be involved in an illegal insider trade in September 2021. Leveraging internal information, that employee bought NFT just before it was featured on the front page of the marketplace, and then sold it right after it soared in price—making a profit of 18.875 ETH in total.

3.14 Conclusion

In this chapter, we discuss the emergence of Non-Fungible Tokens (NFTs) as a means of collecting digital art and investment. We point out that despite the rapid growth and popularity of the NFT markets, they have not received much security compared to other decentralized finance (DeFi) protocols. In this work, we aim to study the dynamics and security issues of this multi-billion dollar NFT ecosystem. We first provide an overview of how the NFT ecosystem works, identify the major actors involved, and analyze the top NFT marketplaces for potential issues that could result in financial losses. Further, we explore the risks posed by external entities, and examine malicious trading behaviors carried out by the users. Additionally, our research has uncovered and quantified various unethical trading practices, including wash trading, shill bidding, and bid shielding, that are occurring in the major NFT marketplaces. This work offers a comprehensive analysis of the NFT ecosystem, leveraging data from blockchain transactions, and NFT marketplaces, and proposes mitigations to improve the security and awareness of NFT marketplaces and users.

Chapter 4

Hybrid Pruning: Towards Precise Pointer and Taint Analysis

Pointer analysis is a fundamental static program analysis technique that computes the set of abstract program objects that a pointer variable may or must point to. Pointer information is an indispensable pre-requisite for various techniques operating across a spectrum of domains, ranging from programming languages, to software engineering, to system security. One such notable client is taint analysis, which determines the set of objects in a program that are affected by external inputs. The analysis is bootstrapped by marking an initial set of objects that can directly be influenced by an external source (*e.g.*, an attacker) as *tainted*. During taint propagation, the taint engine consults the points-to set of the destination operand of a program instruction, and propagates taint labels according to the taint policy, and the taint labels of the source operands. Therefore, an over-approximated points-to set quickly leads to taint explosion, resulting in most of the program objects getting incorrectly tainted. Many static vulnerability detection techniques employ either pointer, or taint analysis, or a combination of both [161]. In order to not miss bugs, these techniques strive to be *sound*, rather than *complete*.

Consequently, such vulnerability detection clients generate numerous false positives. A precise pointer or taint analysis improves the false positive rate of a static vulnerability detector, thereby making the overall result more amenable to manual triaging.

As the size of the target program grows, precise, whole program pointer and taint analyses become prohibitively expensive. Though field, context, or flow sensitivity increases the analysis precision, such an analysis pays the price in terms of the overhead associated with the metadata management, and enumeration of individual field, context, or flow. Oftentimes, the analyses make unsound choices in order to remain scalable, *e.g.*, restricting the exploration within a specific sub-system, or making certain *soundy* assumptions [161] .

We propose *hybrid pruning*—a novel program analysis paradigm that augments the state-of-the-art static analysis techniques with dynamic trace information. Our algorithm improves both the pointer and taint analyses at those program points where static reasoning is imprecise, and precise dynamic information is available. With the recent tide of research in fuzzing, it has become easier to generate high-quality dynamic traces with deeper program penetration. If the dynamic trace is available along a certain program path, our algorithm injects guaranteed, precise yet partial ground truth to aid the static analysis component. Although inherently *unsound* in principle, our strategy transitively improves the analysis at all those program points which were previously using the imprecise static information, thus multiplicating the advantage. However, leveraging a dynamic trace for static analysis is non-trivial, as they operate in two different analysis domains, *e.g.*, concrete instructions and run-time memory allocations *vs.* `SSA`-based `IR` and abstract memory objects. Our approach lifts the dynamic trace to the static domain to make the interleaving possible. Of course, dynamic analysis tools, such as fuzzers, will likely not succeed in exercising all possible program paths. To compensate for the lack of dynamic coverage, we fall back to the conservative static analysis for all other program paths for which a dynamic trace is absent. We demonstrate two different modes of *hybrid pruning* – *opportunistic* ($\mathtt{H_o}$) and *propagation-only* ($\mathtt{H_P}$), and show when one is better than the other depending on the quality of the dynamic trace collected. These two modes operate along a spectrum of *soundness* and *usability.* The improvement in points-to and taint analyses is positively correlated with the dynamic coverage. If the dynamic coverage is moderate, the $\mathtt{H_o}$ mode is preferred. This mode is designed to be more robust against the lack of dynamic information, because it conservatively switches to pure static mode where dynamic information is unavailable. On the other hand, the $\mathtt{H_P}$ mode shows promise when we have high confidence in the quality of dynamic information, as just the dynamic facts are propagated using the static analysis algorithms in this mode.

Our work is motivated by the observation that the static bug detectors are oftentimes notorious in emitting warnings in a volume which far surpasses the triaging ability of the human experts. For example, as on May 7, 2022, Coverity [162], a popular static bug detector, has emitted 47,038 warnings in the Linux kernel version 5.18.0-rc4, of which 9,137 are still outstanding. We envision *hybrid pruning* as a technique to improve the state-of-the-art in the static bug detection. Therefore, to evaluate the applicability of our technique in the real world, we extended DR.CHECKER [161], a purely static bug finder, to make use of *hybrid pruning.* As we anticipated, the precise points-to and taint information indeed reduced the number of false positives, while maintaining a comparable true positive rate. On our evaluation of 12 `CGC` [31] applications, the bug detectors relying on the $\mathtt{H_o}$ mode emit up to 36% less warnings, while the $\mathtt{H_P}$ mode reduces warnings up to 56%. We additionally show that, in spite of reducing significant fraction of warnings, the vulnerability detectors are still able to detect 15 ($\mathtt{H_P}$) and 19 ($\mathtt{H_o}$) out of 20 bugs in the `CGC` [31] and the real-world datasets combined.

$$\frac{\text{p = \&x}}{l_x \in \mathit{PtsTo}(p)}\ \textsc{Address-Of} \qquad \frac{\text{p = q}}{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(q)}\ \textsc{Copy} \qquad \frac{\text{p = *q}}{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(*q)}\ \textsc{Dereference} \qquad \frac{\text{*p = q}}{\mathit{PtsTo}(*p) \supseteq \mathit{PtsTo}(q)}\ \textsc{Assign}$$

Figure 4.1: The premise (highlighted) of an inference rule represents the type of the statement encountered in a program, and the conclusion corresponds to the constraints in SPT.

4.1 Background

In this section, we equip the reader with the background information required to understand our approach.

4.1.1 Flow-sensitive, static points-to analysis

We provide a brief overview of an Andersen-style, flow-sensitive, static points-to (SPT) analysis technique, which we will use later on to demonstrate our hybrid approach. The goal of any static points-to analysis is to determine the set of objects that a given pointer can point to, at any point in the program. Specifically, a points-to analysis answers a membership query *IsPtsTo(p, x)*, which indicates whether a memory object x is in the points-to set of the pointer p. A flow-sensitive points-to analysis computes the points-to set of the pointers according to the control-flow of the program. Given a program, the analysis starts by generating constraints for every pointer according to their usage in the program. The solution to the generated constraints gives the points-to results for all the pointers. A points-to analysis either categorizes, or transforms any program statement into one or more of the statements in Figure 4.1.

Constraint generation. The analysis iterates over the statements in a program, and collects the constraints according to the rules in Figure 4.1, where l_x and *PtsTo(p)* denote the location of the variable x, and the points-to set of the pointer p respectively. The constraints are usually managed by creating a *constraint graph*, where the nodes represent

$$\frac{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(q) \quad l_x \in \mathit{PtsTo}(q)}{l_x \in \mathit{PtsTo}(p)}\ \textsc{Copy} \qquad \frac{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(*q) \quad l_r \in \mathit{PtsTo}(q) \quad l_x \in \mathit{PtsTo}(r)}{l_x \in \mathit{PtsTo}(p)}\ \textsc{Dereference}$$

$$\frac{\mathit{PtsTo}(*p) \supseteq \mathit{PtsTo}(q) \quad l_r \in \mathit{PtsTo}(p) \quad l_x \in \mathit{PtsTo}(q)}{l_x \in \mathit{PtsTo}(r)}\ \textsc{Assign}$$

Figure 4.2: Rules to solve the SPT constraint graph.

pointers or memory objects, and edges represent the constraints.

Constraint solving. Once the constraints are generated, each of the constraints will be solved until a fixed point is reached, *i.e.*, no changes occur to the points-to set of all the pointers. The rules in Figure 4.2 are used to solve the generated constraints.

4.1.2 Static taint tracking

Static Taint Tracking (STT) [163] is a data-flow tracking technique used to track the flow of the tainted data within a program. STT is most commonly used in vulnerability detection, where the program input is tainted, and vulnerabilities are modeled as an usage of unsanitized data in the sensitive operations. For example, a usage of tainted data in an arithmetic operation can cause an integer overflow or an underflow bug. Similarly, an out-of-bounds access bug can occur when tainted data is used as the index in an array. STT consists of the following components:

Taint source. Functions that read an input from the user, or the environment, *e.g.*, `read`, `scanf` are considered as taint sources. The variables into which the data is read are labeled as *tainted*.

Taint propagation. Typically, the result (destination) of an operation is labeled *tainted* if any one of its operands (source) is already tainted, *e.g.*, for a binary operation $r \leftarrow f(a, b)$, taint propagates to r if either a or b is tainted.

An STT requires points-to information to track the flow of tainted data through pointers. To inject taint, the STT must know to which objects the source pointer can point, so that it can taint all those objects. Note that an imprecise pointer analysis could result in over-tainting, resulting in many data elements being incorrectly considered as tainted [164]. In this work, we use the taint propagation rules similar to the ones proposed in DR. CHECKER [161].

4.2 Motivation

4.2.1 Running example

We use the code in Listing 4.1 to explain various aspects of our technique. To generate execution traces, we exercise the program with a test suite. However, the part of the code highlighted in *red* is ***not*** executed in any of the dynamic runs.

Points-to. The `process_buf()` function returns either of its `char` pointer arguments (`res` at Line 10, or `req` at Line 13) depending on the value of `r` (Lines 8 and 11). `c_buff` gets assigned the pointer returned by the `process_buf()` call once at Line 28 (`res_buff`), then again at Line 35 (`greq`), and lastly at Line 47 (`q`).

Taint. At Line 32, the program reads user data into the buffer pointed to by `c_buff`, which, in turn, points to `res_buff`.

Bugs. There are five array indexing operations (Lines 37 – 41, 50). However, the operation at the Line 39 could lead to an out-of-bounds write of `buff`, because `res_buff` gets *tainted* at Line 32 (via `c_buff`). In turn, the index `res_buff[0]` can contain a value greater than the size of `buff` (*i.e.*, 16). Likewise, the `write` at Line 50 can lead to an out-of-bounds write of the buffer pointed by `c_buff` (*i.e.*, `q` from Line 47). The remaining three indexing operations are safe.

4.2.2 Imprecision in vanilla static analysis

Consider an STT technique based on the SPT analysis that we presented in Section 4.1 on our example in Listing 4.1. The call to `read_user_data` taints object ids $\{3, 1, 4, 2\}$ because of the points-to set of `c_buff@28`. At Lines 37, 39, 40, 41, and 50, we are using data from the tainted objects (*i.e.*, $\{3, 1, 4, 2\}$) as indices to write to arrays. Consequently, any static vulnerability detection technique that relies only on the STT information, and checks for the use of tainted data as an array index (unsafe operation) will raise a potential out-of-bounds alert. However, as described in Section 4.2.1, only the buffer pointed to by `res_buff` contains tainted data. Therefore, only the warnings raised at Lines 39 and 50 are true positives. Next, we show how we use dynamic information to improve the precision of static analysis techniques to eliminate these false positives.

```
1  #define BSIZE 512
2  // global object, ID: 1
3  char greq[BSIZE];
4  // global object, ID: 2
5  char gres[BSIZE];
6  char *process_buf(IOLevel r, char *res, char *req) {
7    switch(r) {
8      case IORECV:
9        ...
10       return res;
11     case IOSEND:
12       ...
13       return req;
14   }
15   return NULL;
16 }
17
18 int main(...) {
19   // stack object, ID: 3
20   char req_buff[BSIZE];
21   // stack object, ID: 4
22   char res_buff[BSIZE];
```

```
  // stack object, ID: 5
  char buff[16];
  char *c_buff, *t_buff;
  ...
  // The return value will be res_buff
  c_buff = process_buf(IORECV, res_buff, req_buff);
  ...
  // Read user (tainted) data into the buffer
  // pointed to by c_buff, i.e., res_buff
  read_user_data(c_buff, BSIZE);
  ...
  // The return value will be greq
  c_buff = process_buf(IOSEND, gres, greq);
  ...
  buff[req_buff[0]] = 'R';
  // BUG: Potential out-of-bounds write
  buff[res_buff[0]] = 'S';
  buff[greq[0]] = 'r';
  buff[gres[0]] = 's';
  ...
  if (...) {
    // heap object, ID: 6
    char *q = getenv(...);
    t_buff = c_buff;  // c_buff points to greq here
    c_buff = q;
    ...
    // BUG: Potential out-of-bounds write
    c_buff[res_buff[0]] = 'I';
    ...
  }
  ...
  return 0;
}
```

Listing 4.1: Example program to demonstrate the effectiveness of *hybrid pruning*. The region highlighted in red is ***never*** executed in any of the dynamic runs.

Objects		**Tainted data?**	**Static Taint Tracking (STT)**		
ID	**Name**	**(Ground Truth)**	**Flow-Sens PT**	**H_P-PT**	**H_o-PT**
1	`greq`	✗	✓	✗	✗
2	`gres`	✗	✓	✗	✗
3	`req_buff`	✗	✓	✗	✗
4	`res_buff`	✓	✓	✓	✓
5	`buff`	✗	✗	✗	✗
6	`q`	✗	✗	✗	✗

Table 4.1: Tainted objects (✓: Tainted, ✗: not Tainted) when different points-to analysis techniques are used. The colors green and red represent true positives, and false positives respectively.

Vulnerability Warnings	**Static Taint Tracking (STT)**		
(Ground Truth)	**Flow-Sens PT**	**H_P-PT**	**H_o-PT**
Out-of-bounds write on Line 39	1	1	1
Out-of-bounds write on Line 50	1	0	1
False positives	3	0	0
Total warnings	5	1	2

Table 4.2: Vulnerability warnings of static taint tracking when different points-to analysis techniques are used. The color green represents true positives, and red represents false positives and false negatives respectively.

Pointer	**Dynamic points-to**
`q`	N/A
`buff`	{5}
`req_buff`	{3}
`greq`	{1}
`res_buff`	{4}
`gres`	{2}
`req`	{1,3}
`res`	{2,4}
`return`	{3,1,4,2}
`c_buff@28`	{4}
`c_buff@35`	{2}
`c_buff@47`	N/A

Table 4.3: Dynamic points-to information collected for the example in Listing 4.1. N/A indicates that the code corresponding to the pointer is not executed in any of the dynamic runs.

4.2.3 Precision gain due to *hybrid pruning*

First, we exercise the program either using tests, or by fuzzing, to collect dynamic points-to and taint facts. Then, we augment the static pointer and taint analysis techniques with the recorded dynamic facts in either of the following two ways – *propagation-only* (H_P), or *opportunistic* (H_o).

Propagation-only (H_P). In this mode, the static analysis is first initialized with the recorded dynamic facts. Then, the static pointer and taint analysis algorithms propagate those dynamic facts even to those program points that are not executed dynamically. In other words, the information generated at any program point is derived ***only*** from the dynamic information, but propagated by the static analysis rules. The benefit of the H_P over dynamic-only analysis is that the former compensates for the lack of dynamic information by static propagation of dynamic facts, at the program points where the dynamic information is absent. Greater the dynamic coverage is, more effective the H_P

mode will be in eliminating the spurious points-to and taint sets. The $\mathtt{H_P}$ *hybrid pruning* strategy, when applied to static points-to (SPT) and static taint-tracking (STT) analyses, yields $\mathtt{H_P}$-PT (propagation-only points-to) and $\mathtt{H_P}$-TT (propagation-only taint-tracking) analyses, respectively.

In Listing 4.1, $\mathtt{H_P}$-PT prunes the over-approximated SPT set of `c_buff@28` from $\{3, 1, 4, 2\}$ to $\{4\}$. Consequently, an STT that relies on $\mathtt{H_P}$-PT correctly taints only the object with id 4 (`res_buff`), thus improving the precision of the taint analysis, as shown in Table 4.1 (Column $\mathtt{H_P}$-PT). Furthermore, as shown in Table 4.2 (Column $\mathtt{H_P}$-PT), a static vulnerability detection technique that uses this hybrid taint-tracking emits no false warnings. However, for cases where dynamic information is inadequate, *e.g.*, the points-to information of `c_buff@47` is absent, the $\mathtt{H_P}$ mode might fail to compute certain information. The missing information might introduce false negatives, as shown in Table 4.2 (Column $\mathtt{H_P}$-PT), where using $\mathtt{H_P}$-PT resulted in missing the vulnerability in Line 50 of Listing 4.1.

▶ *Difference between $\mathtt{H_P}$ and classic dynamic analysis.* Since $\mathtt{H_P}$ mode propagates dynamic facts using static algorithms, it essentially compensates for the 'lost' information at certain program points. In Listing 4.1, a purely dynamic approach would compute an empty points-to set for `t_buff@46`, because Line 46 was never executed in any of the dynamic runs. However, Line 35 was dynamically executed, which made `c_buff@35` point to `greq`. That information will be propagated in $\mathtt{H_P}$ mode, resulting in `t_buff@46` correctly pointing to the `greq` buffer.

Opportunistic ($\mathtt{H_o}$). As explained above, the points-to and taint information that the $\mathtt{H_P}$ mode propagates might be incomplete due to lack of dynamic coverage at certain program points – resulting in false negatives. To alleviate this issue, we use the dynamic information in the $\mathtt{H_o}$ mode only at those program points that are executed dynamically.

For all other program points, we use the static information. Opportunistic use of the dynamic facts conservatively preserves the static points-to and taint sets at those program points where the dynamic information is not available. The only difference between the $\mathtt{H_o}$ and the $\mathtt{H_P}$ modes is that the $\mathtt{H_o}$ allows static information to be generated, while the $\mathtt{H_P}$ does not. The $\mathtt{H_o}$ *hybrid pruning* strategy, when applied to static points-to (SPT) and static taint-tracking (STT) analyses, yields $\mathtt{H_o}$-PT (opportunistic points-to) and $\mathtt{H_o}$-TT (opportunistic taint-tracking) analyses, respectively.

In Listing 4.1, though the code highlighted in red is not dynamically executed, $\mathtt{H_o}$-PT infers the points-to relation between `c_buff@47` and object with id 6. Consequently, an STT that relies on $\mathtt{H_o}$-PT correctly taints the relevant buffer, as shown in Table 4.1 (Column $\mathtt{H_o}$-PT). Furthermore, as shown in Table 4.2 (Column $\mathtt{H_o}$-PT), a static vulnerability detection technique that uses this hybrid taint-tracking emits no false warnings, yet discovers both the vulnerabilities.

4.3 Hybrid Pruning

Our technique works in three steps. First, we generate the dynamic facts (Section 4.3.1), *e.g.*, points-to and taint sets, by exercising the program with a test suite, or using fuzzing. In the next phase, which we call *domain re-mapping* (Section 4.3.2), we lift the dynamic facts to the same domain as that of the static ones, so that a unified analysis becomes possible. Finally, we run the static analysis, and inject (Section 4.3.3) the dynamic facts, wherever available, thus eliminating potentially spurious points-to and taint sets at those program points. Note that the precision improvement is **not only** local to the point of injection, but also carried forward to the downstream analysis sites by the static algorithms. For example, “fixing” an over-approximated points-to set progressively taints fewer objects further down the analysis. Finally, we run a number of vulnerability detectors (Section 4.3.4), which uses the hybrid facts to eliminate spurious warnings.

4.3.1 Generation of dynamic facts

During a program's execution, we record **(i)** the allocation and deallocation of program objects, **(ii)** the read and write accesses on those objects, **(iii)** the callsite-based program context of the instructions involved in (i) and (ii), and **(iv)** the arguments of the input API, *e.g.*, `read`. Once this information is collected, we compute the dynamic points-to and taint information corresponding to those memory objects from the recorded trace. Next, we describe how we recover the dynamic facts from the collected trace in detail.

Dynamic context. We keep track of a function's call-stack c at run-time, by emulating a parallel stack updated at every `call` and `ret` instruction. For every instruction I, we compute its dynamic context $\Delta(I) = (c, \tau)$, where c is the call-stack with which I is executed, and τ is an unique identifier for each I.

Memory objects. We maintain the tuple $(sz, rt, \Delta(I))$ for each memory object o allocated, or deallocated by an instruction I, where sz is the size of the object (in bytes), rt is its run-time address, and $\Delta(I)$ being its dynamic context. We extract the size sz of the local and global memory objects from their types. The size of the heap object is extracted from the *size* argument passed to the allocation routines, *e.g.*, `malloc`. Note that, different instances of an object o with the same context $\Delta(I)$ might get created at different points in time in an execution, or even across different executions. We merge the dynamic facts associated with all those instances of an object o, by its context $\Delta(I)$, at the end of trace collection. For each object o created by the same instruction I with the same context $\Delta(I)$, we compute its id $\pi(o) = \{hash(\Delta(I), \tau(I))\}$, which uniquely identifies the object for a given context.

Points-to facts. To compute the points-to sets, we track all the write operations to the program objects. Assume, a memory `write` instruction writes to the address w_d of a memory object $o_d = (sz_d, rt_d, \_)$. If the value being written to is a memory address w_s of an object $o_s = (sz_s, rt_s, \_)$, then we make the offset $(w_d - rt_d)$ of the object o_d point to the offset $(w_s - rt_s)$ of the object o_s. Formally, the updated points-to set $\rho(o_d, w_d - rt_d) = \rho(o_d, w_d - rt_d) \cup (\pi(o_s), w_s - rt_s)$.

Taint facts. We use the same taint sources as that of static taint analysis. However, different from the static case, dynamically we taint the exact number of bytes read by an input API, *e.g.*, a `read(fd, buf, count)` call taints `count` bytes of the buffer `buf`.

4.3.2 Domain re-mapping

Hybrid pruning seeds static analysis algorithms with the dynamic information. Static analysis operates on an intermediate representation (`IR`), and models program memory in terms of abstract objects. However, dynamic analysis executes native CPU instructions, and objects are created at run-time on the program stack, or heap. We use the following two-fold approach to bridge this gap. First, we assign a unique instruction id τ to each `IR` instruction. Additionally, to represent a memory object, we use a unique object id π as discussed earlier. We include both the τ and π in the dynamic events, and the generated dynamic facts. During the static analysis, we re-use the same τ as that of the dynamic analysis, and use identical definition of static context as that of dynamic context $\Delta(I)$. Hence, the static and dynamic object id π evaluates to be the same, for the same object, created in the same context. The *hybrid pruning* leverages this fact to establish the equivalence between a dynamic memory object, and its static counterpart.

$$\frac{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(q) \quad \mathit{dynV}(q) \quad l_x \in \mathit{DynPtsTo}(q)}{l_x \in \mathit{PtsTo}(p)}\ \textsc{DynCopy} \qquad \frac{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(q) \quad \neg\mathit{dynV}(q) \quad l_x \in \mathit{PtsTo}(q)}{l_x \in \mathit{PtsTo}(p)}\ \textsc{ICopy}$$

$$\frac{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(*q) \quad l_r \in \mathit{PtsTo}(q) \quad \mathit{dynV}(r) \quad l_x \in \mathit{DynPtsTo}(r)}{l_x \in \mathit{PtsTo}(p)}\ \textsc{DynDereference}$$

$$\frac{\mathit{PtsTo}(p) \supseteq \mathit{PtsTo}(*q) \quad l_r \in \mathit{PtsTo}(q) \quad \neg\mathit{dynV}(r) \quad l_x \in \mathit{PtsTo}(r)}{l_x \in \mathit{PtsTo}(p)}\ \textsc{IDereference}$$

Figure 4.3: Rules to solve the hybrid constraint graph.

4.3.3 Injection of dynamic facts

We augment both the static pointer and taint analyses with the dynamic information to achieve *hybrid pruning*. For the pointer analysis, we leverage the flow-sensitive analysis from `SVF` [165]. Our static taint analysis engine is flow-, context-, and field-sensitive. In addition to the family of input `API`s, *e.g.*, `scanf`, `gets`, *etc.*, we consider the command-line arguments of the program as the taint sources. The taint analysis is parameterized by the underlying pointer analysis, *i.e.*, while propagating the taint labels, it queries the pointer analysis for the points-to sets of the destination operand of an instruction. Taint sinks are determined by the taint policies of the respective vulnerability detectors. Depending on how we inject dynamic facts during static analysis, we develop two modes of *hybrid pruning* – *propagation-only* and *opportunistic.*

Propagation-only ($\texttt{H}_\texttt{P}$). In this case, we propagate just the dynamic facts using static analysis rules. For the SPT we presented in Section 4.1, we can achieve $\texttt{H}_\texttt{P}$ *hybrid pruning* by **(i)** ***not*** generating any Address-Of constraints, and **(ii)** modifying the Copy and Dereference constraints to consider only the dynamic information. While **(i)** prevents generation of any new static fact, **(ii)** ensures propagation of dynamic facts following the SPT rules. We split the constraint-solving rules in Figure 4.2 depending on the availability of the dynamic information. Specifically, we follow the DynCopy and DynDerefer-

ENCE rules as shown in Figure 4.3, to process the COPY and DEREFERENCE instructions in the $\mathtt{H_P}$ mode. The $dynV(p)$ predicate checks whether the program point corresponding to the pointer p has been dynamically executed. If so, we consider the dynamic points-to set returned by the $DynPtsTo(p)$ predicate. In the $\mathtt{H_P}$ mode of STT, we ignore all the static taint sources. We use just the dynamic taint information for all the dynamically executed instructions. In effect, we consider only those instructions that have been both dynamically executed, and found to be tainted. Due to the space constraint, we refrain from presenting the modified transfer functions for the taint propagation.

Opportunistic ($\mathtt{H_o}$). In this case, we generate new static facts, if dynamic information is unavailable. If the later is available at a program point, it is given priority over its static counterpart. For the SPT we presented in Section 4.1, we can achieve the $\mathtt{H_o}$ *hybrid pruning* by **(i)** generating the ADDRESS-OF constraints, and **(ii)** modifying the COPY and DEREFERENCE constraints to give preference to dynamic information, if available. Otherwise, the constraint solving rules are made to use static information. Policy **(i)** ensures the generation of new static facts, which compensates for the lack of dynamic coverage. In fact, if the dynamic information is available, we use the same constraint-solving rules as in the case of the $\mathtt{H_P}$ mode, while processing the COPY and DEREFERENCE instructions. However, we also introduce two new rules, *viz.*, ICOPY and IDEREFERENCE as shown in Figure 4.3, to deal with those cases when dynamic information is absent. The $\mathtt{H_o}$ strategy falls back to SPT in that case. In the $\mathtt{H_o}$ mode of STT, we enable the static taint sources. Also, we propagate the static taint except when the dynamic information is available at an instruction, it is given priority. In other words, the taint engine never taints an instruction that has been dynamically executed, yet was never tainted.

4.3.4 Vulnerability detection

The vulnerability detectors use the taint information to detect potentially buggy program points. Since, the taint analysis itself is a client of the pointer analysis, the checkers run when both the pointer and taint analyses are over. In our research prototype, we only use detectors capable of finding spatial vulnerabilities, *e.g.*, buffer overflow, out of bounds, *etc.* Temporal bugs, *e.g.*, use after free, double free, *etc.*, are considered out of scope. Specifically, we use the taint-based bug detectors, *i.e.*, *Improper Tainted-Data Use Detector* (`ITDUD`), and *Tainted Loop Bound Detector* (`TLBD`) from the Dr. Checker [161] project. `ITDUD` monitors whether tainted data is used in risky functions *e.g.*, `strcpy`, `memcpy`, *etc.* Where as, `TLBD` checks if the loop bound can possibly be tainted.

4.3.5 Implementation

To generate the dynamic facts, we instrument the program using `LLVM` 7.0 [166]. The static and hybrid analysis engines are based on `SVF` 7.0 [165]. We extended `SVF` to add support for taint analysis, while using its pointer analysis (`fspta`) out-of-the-box. We use the `DataFlowSanitizer` [167] (`DFSan`), a generic dynamic data flow analysis `LLVM` pass, which instruments the program to perform dynamic taint tracking. The `DFSan` also handles memory taint by maintaining a *shadow* memory [168]. Our analysis injects and propagates taint automatically, and collects all the tainted instructions and memory objects. The vulnerability checkers are adapted from the Dr. Checker [161].

4.4 Evaluation

We evaluate the effectiveness of our approach in a downstream security application, *e.g.*, vulnerability detection. We show that using our hybrid points-to and taint analysis we generate fewer false-positives (warnings that are *not* real bugs), while still detecting ***real*** bugs.

4.4.1 Evaluation setup

Dataset Our approach was evaluated on the following two datasets.

`CGC`. The corpus of 246 programs [31] used by DARPA in the Cyber Grand Challenge (`CGC`) [169]. We chose to use `cb-multios` [170], a port of the `CGC` challenge set to Linux x86 by TRAIL OF BITS. `cb-multios` project failed to port five programs to Linux. Moreover, the programs are meant to be compiled in 32-bit, while our `DFSan` based implementation generates only 64-bit binaries owing to the limitation imposed by the shadow memory mechanism. Due to unsupported architecture, 89 programs aborted with an early memory corruption inside the custom heap allocator. From the remaining ones, we randomly sampled 12 programs containing spatial vulnerabilities to include in our dataset.

Real-world. Though `CGC` programs mimic real-world applications both in terms of complexity and functionality, we collected 8 vulnerable versions (Table 4.4) of 4 distinct real-world `GNU` applications containing only *spatial* vulnerabilities from the `CVE` database [171]. We used the test suites available with the respective versions of those utilities to exercise those programs.

Instrumentation. This step was carried out on an Ubuntu 18.04.3 LTS, 64-bit system equipped with an Intel Core i7-4770 (3.40GHz) CPU, and 32GB of memory, under moderate workload.

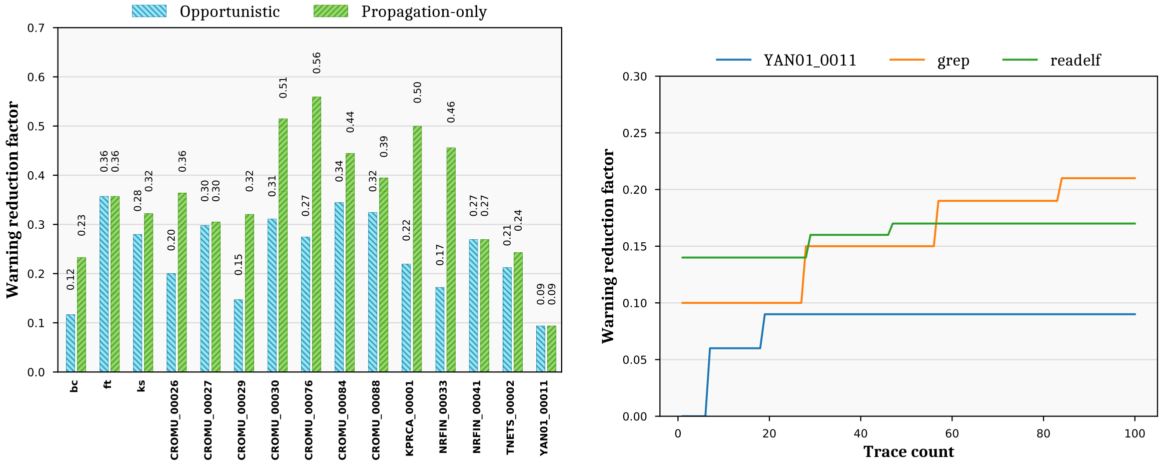
Trace collection. We re-used the same setup from the previous phase. The programs were exercised by their respective test suite. Real-world applications were let to run until they gracefully exit. However, many `CGC` programs being interactive, and menu-driven in nature, they run in a waiting loop until a specific program option is chosen, *e.g.*, sending a `QUIT` command. It is not guaranteed that the test cases will drive the programs to completion. To ensure the convergence of the experiment, we imposed a hard time-limit of 15 seconds per program execution by sending a `SIGTERM` signal, and installed signal handlers to record traces at termination.

Hybrid analysis. We deployed this analysis to a `Celery` [172] cluster consisting of 8 servers with an analysis time-limit of 6 hours per program, per configuration. Each server was equipped with an Intel Xeon E5645 2.40GHz CPU, and 96GB of memory, running Ubuntu 16.04.6 LTS, 64-bit. Despite the time-limit in place, ***none*** of the analyses was observed to hit the limit.

4.4.2 Vulnerability detection

We measured the effectiveness of our pruning strategy in terms of the reduction of warnings due to the following two reasons—**(i)** We were interested to understand if our technique is able to significantly bring down the number of warnings emitted by a static bug detector such that those alarms can be verified by the analysts manually. **(ii)** We had the partial knowledge of the vulnerabilities (ground truth) present in our dataset. In other words, we did not have the knowledge of all the bugs contained in our subjects. Both the sources of building the ground truth—the bugs documented with the `CGC` dataset, and the CVE database records for the vulnerable real-world programs—were incomplete. Therefore, we could only confidently determine the true positives for bugs by associating the warnings to our known bugs. However, a similar strategy would incorrectly flag a warning, which is indeed a bug, as a false positive just because the associated bug report is not present in our (incomplete) ground truth. Establishing a complete ground truth would not only require the involvement of human experts, but also would be hard to scale to all the programs included in our dataset.

Warning Reduction Factor (`WRF`). To measure the effectiveness of *hybrid pruning*, we introduce the notion of *warning reduction factor* (`WRF`), a metric that captures the effect of *hybrid pruning* on emitted warnings, *w.r.t.* the baseline static vulnerability detection technique. An $\texttt{WRF} = 0\%$, the worst-case scenario for our technique, corresponds to no improvement due to *hybrid pruning* over the static analysis. A non-zero `WRF` quantifies the improvement in performance induced by *hybrid pruning*. We define `WRF` as the fraction of warnings that are **not** raised by our hybrid ($\mathtt{H_o}/\mathtt{H_P}$) analysis. Formally, $\texttt{WRF} = (|\omega_{\mathcal{B}}| - |\omega_{\mathcal{H}}|)/|\omega_{\mathcal{B}}|$, where $|\omega_{\mathcal{B}}|$ and $|\omega_{\mathcal{H}}|$ denote the number of warnings reported by the baseline and the hybrid analyses respectively.



(a) Warning reduction by *hybrid pruning* compared to baseline analysis

(b) Warning reduction with dynamic trace

Figure 4.4: Analysis of warning reductions

Results and analysis. In this experiment, we ran the bug detectors (Section 4.3.4) in three different configurations: **(A) static-*only*:** flow-sensitive static points-to + static

Subject	Warnings				
	Static	$\mathtt{H_o}$	Bug Found?	$\mathtt{H_P}$	Bug Found?
`CROMU_00026`	249	199	✓	158	✗
`CROMU_00027`	141	99	✓	98	✓
`CROMU_00029`	261	223	✓	177	✗
`CROMU_00030`	305	210	✓	148	✓
`CROMU_00076`	321	233	✓	141	✗
`CROMU_00084`	700	459	✓	389	✓
`CROMU_00088`	528	357	✓	320	✓
`KPRCA_00001`	209	163	✓	105	✓
`NRFIN_00033`	93	77	✓	51	✓
`NRFIN_00041`	268	196	✓	196	✓
`TNETS_00002`	33	26	✓	25	✓
`YAN01_00011`	32	29	✓	29	✓
`readelf-2.28 (CVE-2017-6969)`	2255	1872	✓	999	✗
`readelf-2.28 (CVE-2017-8398)`	2255	1872	✓	999	✓
`readelf-2.30 (CVE-2018-10372)`	3038	2582	✓	1231	✓
`readelf-2.32 (CVE-2019-14444)`	3061	2663	✓	1176	✓
`readelf-c0e331c (CVE-2017-15996)`	2369	2037	✗	996	✗
`date-15fca2a (CVE-2014-9471)`	581	238	✓	192	✓
`locate-4.2.30 (CVE-2007-2452)`	1038	571	✓	343	✓
`grep-235aad7 (CVE-2012-5667)`	539	426	✓	270	✓

Table 4.4: Warnings emitted, and corresponding bugs (true positives) discovered by bug finders based on pure static, $\mathtt{H_o}$, and $\mathtt{H_P}$ modes of points-to and taint analyses. ✓ and ✗ denote if the bug has been found or missed by an analysis.

taint **(B) $\mathtt{H_o}$-*only*:** flow-sensitive $\mathtt{H_o}$ points-to + $\mathtt{H_o}$ taint, and **(C) $\mathtt{H_P}$-*only*:** flow-sensitive $\mathtt{H_P}$ points-to + $\mathtt{H_P}$ taint. To demonstrate the reduction in the warnings, we evaluated our approach on the `CGC` [31] dataset. The static-*only* configuration, which emits the *most* number of warnings, serves as the baseline for this experiment. Figure 4.4a shows the reduction in the number of warnings when $\mathtt{H_o}$-*only* and $\mathtt{H_P}$-*only* analyses are used. Further we observe that the `WRF` increases as the size (lines of code), and the complexity (*e.g.*, pointer-heavy programs), the number of taint sources, or the dynamic coverage increases. Intuitively, the first three factors make the analysis harder for a static bug detector, thus generating larger number of spurious warnings. The fourth factor, *i.e.*, the dynamic coverage, indeed benefits the hybrid analysis, as we show in Section 4.4.3. The $\mathtt{H_o}$-*only* configuration reduces the warnings up to 36% (`WRF`=0.36), while the reduction in the $\mathtt{H_P}$-*only* configuration is higher, up to 56% (`WRF`=0.56). We argue that this is no worse than any dynamic-*only* analysis system (*e.g.*, fuzzing) which suffers from insufficient coverage. $\mathtt{H}_{\mathtt{P}}$-*only* mode is helpful only when we have high confidence in the completeness of the dynamic information, *e.g.*, the test suite is exhaustive, providing good coverage. If the dynamic coverage is lacking, $\mathtt{H}_{\mathtt{o}}$-*only* mode is preferred.

This study reinforces the trade-off [173] between *usability* and *soundness*. We envision our bug detection system to be used in practice in either of these two modes: **(a) Conservative:** When an analyst chooses to minimize the likelihood of missing bugs, but is ready to tolerate a reduced reduction in the warnings; $\mathtt{H}_{\mathtt{o}}$-*only* mode is helpful. **(b) Priority:** When an analyst prioritizes finding the *most* number of bugs in a small time window, thus requiring a significant reduction in spurious warnings; $\mathtt{H}_{\mathtt{P}}$-*only* mode is a perfect fit.

While cutting down the number of warnings is desirable, it is not sufficient because of the potential risk of missing the true bugs. To evaluate the impact of *hybrid pruning* on the bug detection capability of the static bug detectors, we ran the same on both the `CGC` [31] and the real-world datasets. Table 4.4 summarizes the bugs discovered by the bug detectors while running in the $\mathtt{H}_{\mathtt{P}}$-*only* and $\mathtt{H}_{\mathtt{o}}$-*only* configurations. While the former is found to miss five bugs, the later misses just one bug. Intuitively, though insufficient dynamic coverage exhibits greater warning reduction in the $\mathtt{H}_{\mathtt{P}}$-mode, it misses more bugs than the $\mathtt{H}_{\mathtt{o}}$-mode, which compensates for the lack of dynamic coverage, by design. Please note that, even $\mathtt{H}_{\mathtt{o}}$-mode can also miss true bugs in some cases. We discuss that in Section 4.5.

Hence, we show that *hybrid pruning* enable scalable and efficient bug triaging by cutting down on false alarms while retaining comparable true-positive rate.

4.4.3 Effect of dynamic trace

An important aspect to consider is how the quantity of dynamic information available affects the overall performance of *hybrid pruning*. We conducted this experiment on three subjects, *i.e.*, `YAN01_00011`, `grep` and `readelf` in $\mathtt{H}_{\mathtt{o}}$mode; where we gradually inject more dynamic traces into our analysis system. We use fuzzing as an inexpensive way of trace generation, and randomly pick 100 traces. Every time a new trace is introduced, we continuously monitor the performance of our analysis system in terms of warning (`WRF`) reductions. With more traces being made available, pointer analysis improves, as additional dynamic information yields new points-to sets not discovered before. Moreover, taint analysis improves due to the combined improvement in the points-to sets, as well as the reduction in the spurious static taint sets. Since the bug detectors consume both the pointer and the taint information, the number of warnings reduces over time. Initially, the `WRF` increases, and then becomes stable at the point when the dynamic coverage saturates. We observe that the performance of *hybrid pruning* is positively correlated with the amount and the quality (coverage) of the dynamic trace. We present in Figure 4.4b. Specifically, for every subject, the corresponding line gradient in Figure 4.4b represents the correlation of WRF with the trace count, *e.g.*, gradient increases mean WRF increases as we add more traces.

Gradient increases. Points-to result improves when additional dynamic information yields new points-to sets that has dynamically never been seen before by the analysis. Also, taint can improve either due to more precise points-to sets, or additional dynamic taint information overriding its static counterpart at newer program points. Warning improves as it is positively correlated to the improvement of either or both the factors.

Gradient unchanged. Neither points-to, nor taint improves. Typically, it is the case when multiple traces exercise the same program path.

Gradient decreases. Increased dynamic information can discover more target objects pointed to by the same pointer; thereby increasing the size of its points-to set. Similarly, extensive dynamic information available at the same program point can newly *taint* an object which was found not to be *tainted* in prior runs.

4.5 Limitations and Discussion

Potential false negatives. Our pruning strategy is context-insensitive, meaning that the different call contexts of the same callee method are indistinguishable from each other.

```
 1  void square(int* p) {
 2      *p = (*p) * (*p);    // Unsafe binary operation
 3  }
 4
 5  int main() {
 6      int n, c = 50, i;
 7      scanf("%d", &i);
 8      if (i < 100) {
 9          scanf("%d", &n);     // Tainted input
10          square(&n);
11      } else
12          square(&c);
13      return 0;
14  }
```

Listing 4.2: False negative of *hybrid pruning*. Instructions in **green** are dynamically executed while the **red** ones are not.

In Listing 4.2, `square` is being called from two different contexts at Line 10 and Line 12, making p point to $\{n, c\}$. The tainted input n can flow to the multiplication operation at Line 2, if and only if i is less than 100. However, assume that the program is exercised ***only*** with test cases having i greater than 100. Therefore, in all the dynamic runs, the constant c is passed to the `square` call at Line 12, which establishes the dynamic points-to relation $p \rightarrow c$. During *hybrid pruning*, when the algorithm evaluates the points-to set of p due to the call at Line 10, it will find that the instructions of `square` have already been dynamically visited, albeit from a different context (Line 12). The context-insensitive pruning strategy disregards the difference in call-sites. At this point, the dynamic points-to set will be given preference, and consequently the static points-to relation $p \rightarrow n$ gets killed. Due to the missing points-to relation, the taint engine will no longer propagate the taint to the multiplication operation at Line 2. In turn, the `ITDUD` vulnerability detector will fail to detect the potentially unsafe binary operation. To summarize, the context-insensitive pruning strategy can lead to false negatives in both the pointer and taint analyses, as well the vulnerability detection. As we show in Section 4.4, the performance of *hybrid pruning* is positively correlated with the quantity, and the quality (coverage) of the available dynamic trace.

Theoretical limitation. To detect temporal bugs, *e.g.*, use-after-free (UAF), double free, *etc.*, a bug detector needs to have both the *reachability* (if the attacker can trigger the events), and the *timing* (if the attacker can control the sequence of events) information. Therefore, the taint information alone is not enough in order to detect this kind of bugs. However, such a bug detector could still benefit from the precise pointer information to infer if different events, *e.g.*, *use*, *free*, *etc.*, are operating on the same program objects. Hence, how the hybrid points-to information improves the discovery of the temporal bugs could be an interesting research direction to explore.

Applicability to other analyses. Since *hybrid pruning* is inherently unsound, it is the best fit for applications where *soundness* is not a strict necessity, for example, in static vulnerability detection, limited cases of call-graph and control-flow graph construction, dynamic symbolic execution, *etc.* Indirect call resolution is a challenging problem—a purely static pointer analysis is likely to miss potential targets unless it is configured to be 'overly' conservative, in which case, it may become unusable. Hybrid pruning can indeed be effective, because it can restrict such a pointer to a smaller set of interesting targets.

4.6 Related Work

In this section we will discuss state-of-the-art techniques related to our work.

Pointer analysis: Pointer analysis is a fundamental program analysis technique with a very rich literature [174–177], and wide applications [178,179]. Steensgaard *et. al.* [176] provides a linear time algorithm based on type inference techniques for pointer analysis. Anderson inclusion-based pointer analysis is another important milestone for pointer analysis which provides good precision compared to Steensgaard *et. al.* with an acceptable performance overhead [177]. Yulei *et. al.*perform value-flow, and pointer analysis in an iterative manner to improve the precision of both [165]. Pointer analysis techniques are designed to be sound as they are mostly used in compiler optimization. However, there are other clients of pointer analysis that does not have this requirement. Vulnerability detection is one such client where less false positives [180], and more precision is required. There are few unsound pointer analysis techniques tailored for bug detection [181–183]. Similarly, speculative execution is one such client where the occasional lost of soundness is acceptable [184]. In order to achieve precision, one can also use dynamic analysis which is precise, but can never be sound. Marcus *et. al.*proposes a technique to compute pointer analysis results dynamically, which are called dynamic points-to results [185,186]. They also show that the static pointer analysis results are an order of magnitude imprecise than dynamic points-to results. Another work shows how the dynamic points-to results can be used for program slicing [187]. Additionally, David *et. al.*integrates pointer analysis with Dynamic symbolic execution to increase the precision of pointer analysis [188]. However, dynamic information heavily relies on the tests, and can never be sound. In this work, we explore the possibility of augmenting the static pointer analysis—which is imprecise but sound with dynamic points-to—which are precise but unsound. We then show how this can be used to increase the precision of vulnerability detection techniques.

Taint analysis: Taint tracking is a data flow tracking technique to track the effect of user data at various program points [163]. Static taint tracking [189] requires a precise pointer analysis, else it usually ends up with Taint explosion [164], tainting all program data. Consequently, almost all the static taint tracking techniques are developed for Java [189] and other strongly typed languages where the pointer analysis results are relatively precise. Dynamic taint tracking(`DTT`) [163,190] is usually performed by instrumenting program instructions [190], resulting in memory and run-time overhead. Though several techniques have been developed to improve the run-time overhead; it still suffers from the lack of dynamic coverage [191–194].

Vulnerability detection. Nevertheless imprecise, the importance of static analysis in vulnerability detection is undeniable. A large body of work on the static detection of vulnerabilities in C/C++ programs has evolved over the last two decades. Engler et al.first explored this domain using various static analysis techniques [195–197]. Other techniques target only specific classes of vulnerabilities, such as, buffer overflows [198–200], memory leaks [201], integer anomalies [202,203], and format string errors [204]. However, as the complexity of software grows, these techniques either do not scale, or incur a large number of false positives.

The key motivation behind this work is to bring the best of both the worlds together,*i.e.*, the *scalability* offered by the static analysis, and the *precision* guaranteed by the dynamic analysis. We attempt to combine both in a novel way, such that, we can draw on the strengths of each. There exists previous attempts that combine static and dynamic analysis for various applications [205–211]. Tapti *et. al.*combines static analysis with dynamic data flow tracking (DFT) to increase the the precision of pointer analysis [212]. They used this precise pointer analysis to protect memory disclosure, and transient execution attacks. Other techniques have aimed to improve vulnerability detection as a downstream client, *e.g.*, using dynamic analysis to verify the results of static analysis, guiding fuzzing through static program analysis, using static analysis to localize program faults in untested code from fuzzer generated crash, *etc.* [205–208]. However, none of them combine the static and dynamic analysis in an interleaved way to improve the points-to, and taint analysis—which is further used in vulnerability detection to reduce the false warnings. To our knowledge, we are the first to explore this direction.

4.7 Conclusion

In this chapter, we introduce a new prgroam analysis technique called *hybrid pruning* that enhances the precision of static analysis techniques like pointer and taint analysis by using dynamic information extracted from a program's runtime trace. These static-only analyses often sacrifice precision for scalability when dealing with large programs. Our hybrid technique injects accurate information from a program's dynamic trace into the static analysis system, thus improving its precision. While doing so, we also tackle the challenge of combining static and dynamic analyses, which operate in two different analysis domains. The paper demonstrates the usefulness of this approach by reducing false positives emitted by a static vulnerability detector. While a vanilla static vulnerability detector emits a large number of alarms, thereby making it challenging for a manual analysis to review the legitimacy of all the alarms, our hybrid system cuts down the number of warnings significantly, thus making the task easy for an analyst. The hybrid pruning technique shows promise in improving static bug detection and has been evaluated on real-world applications.

Chapter 5

Conclusion

In this dissertation, I presented novel approaches of finding attacks and vulnerabilities in two major types of critical systems. For NFT marketplace, which is a business-critical system, I presented our findings on past attacks and abuses, along with data analysis techniques and mathematical models to spot similar future attacks. For operating system (OS) kernel and application software, which are security-critical systems, I presented novel dynamic and hybrid analysis techniques, respectively, to discover latent bugs.

First, we focused on the interaction between peripheral devices and respective kernel drivers, which can be complex, and hence writing correct device driver software is hard. Unfortunately, as it has been recently demonstrated, vulnerabilities in wireless communication peripherals and corresponding drivers can be exploited to achieve remote kernel code execution without invoking a single system call. Nonetheless, no versatile framework existed that could analyze this interaction domain. We present PeriScope, a generic probing framework that addresses the specific analysis needs of the two peripheral interface mechanisms MMIO and DMA. Our fuzzing component PeriFuzz builds upon this framework and can help the end user find bugs in device drivers reachable from a compromised device; uniquely, PeriFuzz can expose double-fetch bugs by fuzzing overlapping fetches, and by warning about overlapping fetches that occurred before a driver crash. Using these tools, we found 15 unique vulnerabilities in the Wi-Fi drivers of two flagship Android smartphones, including 9 previously unknown ones.

Then, we conducted the first systematic study of the emerging NFT ecosystem on 8 top NFT marketplaces (NFTM). To start with, we perform a large-scale data collection from various sources, *viz.*, Ethereum mainnet, NFTM websites, and their documentation. We then compile a comprehensive list of design weaknesses originating from the NFTMs and external entities, which often lead to financial consequences. Further, we develop models to detect common trading malpractices, and quantify their prevalence in these marketplaces.

Finally, we introduce *hybrid pruning*, a technique to improve the precision of static points-to and taint analyses by combining dynamic information collected from program's run-time trace. We propose two different modes of operation, *viz.*, opportunistic and propagation-only, whose applicability is decided by the amount of dynamic information available. Our in-depth evaluation demonstrates both significant improvement in the precision of the points-to sets, and the reduction of the taint sets. When static vulnerability detection is used as a client of the improved pointer and taint analyses, the former is able to find 19 out 20 bugs in `CGC` and real-world software, where as cutting down 21% of the false warnings—making the analysis outcome more amenable to manual triaging.

Bibliography

[1] M. Hinchey and L. Coyle, *Evolving critical systems: A research agenda for computer-based systems*, in *IEEE International Conference and Workshops on Engineering of Computer Based Systems*, 2010.

[2] "Dune analytics—opensea." `https://dune.xyz/rchen8/opensea`.

[3] "How nft giant opensea's $3 billion month compares to amazon, ebay and etsy." `https://decrypt.co/79789/opensea-3b-month-ethereum-nft-sales-amazon-ebay-etsy`.

[4] "Dappradar." `https://dappradar.com`.

[5] M. Nadini, L. Alessandretti, F. D. Giacinto, M. Martino, L. M. Aiello, and A. Baronchelli, *Mapping the nft revolution: market trends, trade networks, and visual features*, *Scientific Reports* (2021).

[6] "The 15 most expensive nfts ever sold." `https://decrypt.co/62898/most-expensive-nfts-ever-sold`.

[7] "Etherscan." `https://etherscan.io/`, 2018.

[8] S. Eskandari, S. Moosavi, and J. Clark, *Sok: Transparent dishonesty: Front-running attacks on blockchain*, in *Proc. Financial Cryptography and Data Security*, 2020.

[9] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, *Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges*, in *Proc. IEEE Symposium on Security and Privacy*, 2020.

[10] L. Zhou, K. Qin, C. F. Torres, D. Le, and A. Gervais, *High-frequency trading on decentralized on-chain exchanges*, in *Proc. IEEE Symposium on Security and Privacy*, 2020.

[11] K. Qin, L. Zhou, and A. Gervais, *Quantifying blockchain extractable value: How dark is the forest?*, *ArXiv* **abs/2101.05511** (2021).

[12] K. Qin, L. Zhou, B. Livshits, and A. Gervais, *Attacking the defi ecosystem with flash loans for fun and profit*, in *Proc. Financial Cryptography and Data Security*, 2021.

[13] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, *On the just-in-time discovery of profit-generating transactions in defi protocols*, in *Proc. IEEE Symposium on Security and Privacy*, 2021.

[14] J. Kamps and B. Kleinberg, *To the moon: defining and detecting cryptocurrency pump-and-dumps*, *Crime Science* **7** (Nov, 2018) 18.

[15] N. Gandal, J. Hamrick, T. Moore, and T. Oberman, *Price manipulation in the bitcoin ecosystem*, *Journal of Monetary Economics* **95** (01, 2018).

[16] J. Xu and B. Livshits, *The anatomy of a cryptocurrency pump-and-dump scheme*, in *Proc. USENIX Security Symposium*, 2019.

[17] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, *Securify: Practical security analysis of smart contracts*, in *Proc. Conference on Computer and Communications Security*, 2018.

[18] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, *Madmax: surviving out-of-gas conditions in ethereum smart contracts*, in *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2018.

[19] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, *ZEUS: analyzing safety of smart contracts*, in *Proc. The Network and Distributed System Security Symposium*, 2018.

[20] "Mythril." `https://github.com/ConsenSys/mythril`.

[21] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, *Making smart contracts smarter*, in *Proc. Conference on Computer and Communications Security*, 2016.

[22] J. Frank, C. Aschermann, and T. Holz, *ETHBMC: A bounded model checker for smart contracts*, in *Proc. USENIX Security Symposium*, 2020.

[23] "Manticore.." `https://github.com/trailofbits/manticore/`, 2016.

[24] B. Jiang, Y. Liu, and W. K. Chan, *Contractfuzzer: fuzzing smart contracts for vulnerability detection*, in *Proc. International Conference on Automated Software Engineering*, 2018.

[25] "Echidna." `https://github.com/crytic/echidna`. [accessed 07/27/2020].

[26] T. Nguyen, L. Pham, J. Sun, Y. Lin, and M. Q. Tran, *sfuzz: An efficient adaptive fuzzer for solidity smart contracts*, in *Proc. International Conference on Software Engineering*, 2020.

[27] P. Bose, D. Das, Y. Chen, Y. Feng, and C. K. G. Vigna, *Sailfish: vetting smart contract state-inconsistency bugs in seconds*, in *Proc. IEEE Symposium on Security and Privacy*, 2022.

[28] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, *TXSPECTOR: Uncovering attacks in ethereum from transactions*, in *Proc. USENIX Security Symposium*, 2020.

[29] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, *Online detection of effectively callback free objects with applications to smart contracts*, in *Proc. Symposium on Principles of Programming Languages*, 2018.

[30] "The dao attack.." `https://www.coindesk.com/understanding-dao-hack-journalists`, 2016. [accessed 04/26/2020].

[31] B. Caswell, "Cyber grand challenge corpus."

[32] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain, *Can you still trust your network card?*, *CanSecWest* (2010).

[33] R.-P. Weinmann, *All your baseband are belong to us*, *DeepSec* (2010).

[34] N. Artenstein, *BroadPwn: Remotely compromising Android and iOS via a bug in Broadcom's Wi-Fi chipsets*, *Black Hat USA* (2017).

[35] G. Beniamini, *Over the air - vol. 2, pt. 2: Exploiting the Wi-Fi stack on Apple devices*, 2017.

[36] G. Beniamini, *Over the air: Exploiting Broadcom's Wi-Fi stack (part 1)*, 2017.

[37] A. Cama, *A walk with Shannon: A walkthrough of a pwn2own baseband exploit*, *OPCDE Kenya* (2018).

[38] G. Beniamini, *Over the air: Exploiting Broadcom's Wi-Fi stack (part 2)*, 2017.

[39] G. Beniamini, *Over the air - vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices*, 2017.

[40] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, *Precise and scalable detection of double-fetch bugs in OS kernels*, in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[41] F. J. Serna, *MS08-061 : The case of the kernel mode double-fetch*, 2008.

[42] M. Jurczyk and G. Coldwind, *Identifying and exploiting Windows kernel race conditions via memory access patterns*, .

[43] S. Keil and C. Kolbitsch, *Stateful fuzzing of wireless device drivers in an emulated environment*, *Black Hat Japan* (2007).

[44] M. Mendonça and N. Neves, *Fuzzing Wi-Fi drivers to locate security vulnerabilities*, in *Proceedings of the European Dependable Computing Conference (EDCC)*, 2008.

[45] M. Jodeit and M. Johns, *USB device drivers: A stepping stone into your kernel*, in *Proceedings of the European Conference on Computer Network Defense (EC2ND)*, 2010.

[46] S. Schumilo, R. Spenneberg, and H. Schwartke, *Don't trust your USB! how to find bugs in USB device drivers*, *Black Hat Europe* (2014).

[47] V. Kuznetsov, V. Chipounov, and G. Candea, *Testing closed-source binary device drivers with DDT*, in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[48] V. Chipounov and G. Candea, *Reverse engineering of binary device drivers with RevNIC*, in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2010.

[49] M. J. Renzelmann, A. Kadav, and M. M. Swift, *SymDrive: Testing drivers without devices*, in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[50] J. Patrick-Evans, L. Cavallaro, and J. Kinder, *POTUS: probing off-the-shelf USB drivers with symbolic fault injection*, in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[51] F. Bellard, *QEMU, a fast and portable dynamic translator*, in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[52] V. Chipounov, V. Kuznetsov, and G. Candea, *S2E: A platform for in-vivo multi-path analysis of software systems*, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[53] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, *CollAFL: Path sensitive fuzzing*, in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[54] P. Chen and H. Chen, *Angora: Efficient fuzzing by principled search*, in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[55] M. Böhme, V.-T. Pham, and A. Roychoudhury, *Coverage-based greybox fuzzing as markov chain*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[56] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, *Directed greybox fuzzing*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[57] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, *Hawkeye: Towards a desired directed grey-box fuzzer*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[58] M. Schulz, D. Wegemer, and M. Hollick, *The Nexmon firmware analysis and modification framework: Empowering researchers to enhance Wi-Fi devices*, *Computer Communications* (2018).

[59] "Facedancer11."

[60] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, *et. al.*, *AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares*, in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[61] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, *Charm: Facilitating dynamic analysis of device drivers of mobile systems*, in *Proceedings of the USENIX Security Symposium*, 2018.

[62] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, *What you corrupt is not what you crash: Challenges in fuzzing embedded devices*, in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[63] D. Aumaitre and C. Devine, *Subverting Windows 7 x64 kernel with DMA attacks*, *HITBSecConf Amsterdam* (2010).

[64] P. Stewin and I. Bystrov, *Understanding DMA malware*, in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[65] A. Markuze, A. Morrison, and D. Tsafrir, *True IOMMU protection from DMA attacks: When copy is faster than zero copy*, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[66] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafrir, *DAMN: Overhead-free IOMMU protection for networking*, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[67] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, *kAFL: Hardware-assisted feedback fuzzing for OS kernels*, in *Proceedings of the USENIX Security Symposium*, 2017.

[68] J. Hertz and T. Newsham, *A Linux system call fuzzer using TriforceAFL*, 2016.

[69] V. Nossum and Q. Casasnovas, *Filesystem fuzzing with american fuzzy lop*, *Vault* (2016).

[70] M. Zalewski, *American fuzzy lop*, 2018.

[71] D. Vyukov, *kernel: add kcov code coverage*, 2016.

[72] Q. Casasnovas, *[patch] kcov: add AFL-style tracing*, 2016.

[73] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, *DIFUZE: Interface aware fuzzing for kernel drivers*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[74] D. Vyukov, *syzkaller - kernel fuzzer*, 2018.

[75] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, *Recovering device drivers*, in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[76] A. Kadav, M. J. Renzelmann, and M. M. Swift, *Fine-grained fault tolerance using device checkpoints*, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[77] H. Han and S. K. Cha, *IMF: Inferred model-based fuzzer*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[78] S. Pailoor, A. Aday, and S. Jana, *Moonshine: Optimizing OS fuzzer seed selection with trace distillation*, in *Proceedings of the USENIX Security Symposium*, 2018.

[79] *ProtoFuzz: A protobuf fuzzer*, 2016.

[80] A. Konovalov and D. Vyukov, *KernelAddressSanitizer (KASan): a fast memory error detector for the Linux kernel*, *LinuxCon North America* (2015).

[81] A. Ryabinin, *UBSan: run-time undefined behavior sanity checker*, 2014.

[82] D. J. Tian, A. Bates, and K. Butler, *Defending against malicious USB firmware with GoodUSB*, in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[83] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish, *Defending against malicious peripherals with Cinch*, in *Proceedings of the USENIX Security Symposium*, 2016.

[84] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor, *Making USB great again with USBFILTER*, in *Proceedings of the USENIX Security Symposium*, 2016.

[85] P. Chubb, *Linux kernel infrastructure for user-level device drivers*, in *Linux Conference*, 2004.

[86] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser, *User-level device drivers: Achieved performance*, *Journal of Computer Science and Technology* **20** (2005), no. 5 654–664.

[87] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, *The design and implementation of microdrivers*, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[88] S. Boyd-Wickizer and N. Zeldovich, *Tolerating malicious device drivers in Linux*, in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[89] *BlueZ: Official Linux Bluetooth protocol stack*, 2018.

[90] *Fluoride Bluetooth stack*, 2018.

[91] *BlueBorne vulnerabilities*, 2017.

[92] J. Pan, G. Yan, and X. Fan, *Digtool: A virtualization-based framework for detecting kernel vulnerabilities*, in *Proceedings of the USENIX Security Symposium*, 2017.

[93] I. Beer, *pwn4fun spring 2014 - Safari - part II*, 2014.

[94] *Project Triforce: Run AFL on everything!*, 2016.

[95] *Trinity: Linux system call fuzzer*, 2018.

[96] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, *Razzer: Finding kernel race bugs through fuzzing*, in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019. To appear.

[97] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *BootStomp: On the security of bootloaders in mobile devices*, in *Proceedings of the USENIX Security Symposium*, 2017.

[98] "Using the Linux kernel Tracepoints."

[99] *ftrace - function tracer*, 2008.

[100] *Kernel probes (Kprobes)*, 2004.

[101] B. Gregg, *Linux extended BPF (eBPF) tracing tools*, 2018.

[102] "LTTng."

[103] "SystemTap."

[104] "ktap: A lightweight script-based dynamic tracing tool for Linux."

[105] R. J. Moore, *A universal dynamic trace for Linux and other operating systems*, in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2001.

[106] *Memory mapped I/O trace*, 2014.

[107] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, *Thorough static analysis of device drivers*, in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2006.

[108] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, *How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel*, in *Proceedings of the USENIX Security Symposium*, 2017.

[109] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, *Dr. Checker: A soundy analysis for Linux kernel drivers*, in *Proceedings of the USENIX Security Symposium*, 2017.

[110] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, *K-Miner: Uncovering memory corruption in Linux*, in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[111] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, *Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features*, in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[112] F. Wilhelm, *Xenpwn: Breaking paravirtualized devices*, *Black Hat USA* (2016).

[113] "Now postage stamps are getting the nft treatment." `https://decrypt.co/61963/now-postage-stamps-are-getting-the-nft-treatment`.

[114] "Usps certifies casemail as first blockchain generated epostage." `https://www.prnewswire.com/news-releases/usps-certifies-casemail-as-first-blockchain-generated-epostage-301267842.html`.

[115] "You can now buy gold-backed nfts with the mining carbon footprint offset." `https://cointelegraph.com/news/you-can-now-buy-gold-backed-nfts-with-the-mining-carbon-footprint-offset`.

[116] "Real estate tokenization." `https://www.blockchainappfactory.com/real-estate-tokenization`.

[117] "Flipkick." `https://www.flipkick.io`.

[118] Q. Wang, R. Li, Q. Wang, and S. Chen, *Non-fungible token (nft): Overview, evaluation, opportunities and challenges*, *arxiv* **abs/2105.07447** (2021).

[119] "Fake banksy nft sold through artist's website for £244k." `https://www.bbc.com/news/technology-58399338`.

[120] "Nft mania is here, so are the scammers." `https://www.theverge.com/2021/3/20/22334527/nft-scams-artists-opensea-rarible-marble-cards-fraud-art`.

[121] "Social engineering nft users through unauthorized support channel." `https://www.theverge.com/22683766/nft-scams-theft-social-engineering-opensea-community-recovery`.

[122] "Solana nft project accused of rug pull after lil uzi deletes tweets." `https://cryptobriefing.com/solana-nft-project-accused-of-rug-pull-after-lil-uzi-deletes-tweets`.

[123] F. Vogelsteller and V. Buterin, "Eip-20: Erc-20 token standard, ethereum improvement proposals, no. 20." `https://eips.ethereum.org/EIPS/eip-20`. 2021/07/11.

[124] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "Eip-721: Erc-721 non-fungible token standard, ethereum improvement proposals, no. 721." `https://eips.ethereum.org/EIPS/eip-721`.

[125] "Ethereum name service (ens) as nft." `https://docs.ens.domains/dapp-developer-guide/ens-as-nft`.

[126] "Cryptokitties." `https://www.cryptokitties.co`.

[127] "Interplanetary file system (ipfs)." `https://ipfs.io`.

[128] "Signing and verifying ethereum signatures." `https://yos.io/2018/11/16/ethereum-signatures`.

[129] "Mastering ethereum." `https://www.oreilly.com/library/view/mastering-ethereum/9781491971932/ch04.html`.

[130] "Opensea." `https://opensea.io`.

[131] "Axie infinity." `https://axieinfinity.com`.

[132] "Cryptopunks." `https://www.larvalabs.com/cryptopunks`.

[133] "Rarible." `https://https://rarible.com`.

[134] "Superrare." `https://superrare.com`.

[135] "Sorare." `https://sorare.com`.

[136] "Foundation." `https://foundation.app`.

[137] "Nifty." `https://niftygateway.com`.

[138] K. Li, J. Chen, X. Liu, Y. Tang, X. Wang, and X. Luo, *As strong as its weakest link: How to break blockchain dapps at rpc service*, in *Proc. The Network and Distributed System Security Symposium*, 2021.

[139] "Coingecko." `https://www.coingecko.com`.

[140] "The art industry and u.s. policies that undermine sanctions." `https://www.hsgac.senate.gov/imo/media/doc/2020-07-29%20PSI%20Staff%20Report%20-%20The%20Art%20Industry%20and%20U.S.%20Policies%20that%20Undermine%20Sanctions.pdf`.

[141] "Hackers stole nfts from nifty gateway users." `https://www.theverge.com/2021/3/15/22331818/nifty-gateway-hack-steal-nfts-credit-card`.

[142] "How do i get a blue checkmark?." `https://support.opensea.io/hc/en-us/articles/360063519133-How-do-I-get-a-blue-checkmark-`.

[143] "Perpetual image hash." `http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html`.

[144] "Imagehash." `https://github.com/JohannesBuchner/imagehash`.

[145] "Behaviors in the nft ecosystem that we hope will decrease in 2020." `https://nonfungible.com/blog/bad-behaviors-nft-blockchain`.

[146] “What is ”wash trading” and why is it negative for non-fungible tokens?.” `https://nonfungible.com/blog/wash-trading-and-why-its-negative-for-non-fungible-tokens`.

[147] “The specter of shill bidding around nfts - tokensmart nft humpday report 15.” `https://nft.substack.com/p/the-specter-of-shill-bidding-around`.

[148] “More rari tokens rewarded than the actual art purchase.” `https://twitter.com/bergleeuw62/status/1293502649308979200?utm_source=nonfungible`.

[149] “Decentraland.” `https://market.decentraland.org/`.

[150] F. Victor and A. M. Weintraud, *Detecting and quantifying wash trading on decentralized cryptocurrency exchanges*, in *Proc. The Web Conference*, pp. 23–32, 2021.

[151] “Bitmix.” `https://bitmix.online`.

[152] “Eth mixer.” `https://eth-mixer.com`.

[153] “Tornado cash.” `https://tornado.cash`.

[154] J. Trevathan and W. Read, *Detecting Shill Bidding in Online English Auctions*, pp. 446–470. IGI Global, 01, 2008.

[155] “Cryptopunk #7523 sale tweet.” `https://twitter.com/Sothebys/status/1402996062474760193`.

[156] “Cryptopunk #7523 sale tweet.” `https://twitter.com/sillytuna/status/1402997178767790082`.

[157] “Digital scarcity.” `https://policyreview.info/glossary/digital-scarcity`.

[158] “Nft art finance.” `https://www.nft-art.finance`.

[159] “Generalized ethereum frontrunners.” `https://blog.kleros.io/generalized-ethereum-frontrunners-an-implementation-and-a-cheat`.

[160] “Did an “art heist” just happen on an ethereum cryptopunks nft?.” `https://cryptoslate.com/did-an-art-heist-just-happen-on-an-ethereum-cryptopunks-nft`.

[161] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, *DR. CHECKER: A soundy analysis for linux kernel drivers*, in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 1007–1024, USENIX Association, 2017.

[162] “Coverity linux scan.” `https://scan.coverity.com/projects/linux`.

[163] E. J. Schwartz, T. Avgerinos, and D. Brumley, *All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)*, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[164] A. Slowinska and H. Bos, *Pointless tainting?: evaluating the practicality of pointer tainting*, in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 61–74, ACM, 2009.

[165] Y. Sui and J. Xue, *Svf: Interprocedural static value-flow analysis in llvm*, in *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[166] “The llvm compiler infrastructure.” `https://llvm.org`.

[167] “Llvm dataflowsanitizer pass.” `https://clang.llvm.org/docs/DataFlowSanitizer.html`.

[168] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, *Addresssanitizer: A fast address sanity checker*, in *USENIX ATC*, 2012.

[169] “Darpa cyber grand challenge.” `https://www.darpa.mil/program/cyber-grand-challenge`.

[170] T. of Bits, “Darpa challenge binaries on linux, osx, and windows.” `https://github.com/trailofbits/cb-multios`, 2016.

[171] “Common vulnerabilities and exposures.” `https://cve.mitre.org`.

[172] “Celery: Distributed task queue.” `http://www.celeryproject.org`.

[173] Y. Xie, M. Naik, B. Hackett, and A. Aiken, *Soundness and its role in bug detection systems*, in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[174] Y. Smaragdakis, G. Balatsouras, *et. al.*, *Pointer analysis*, *Foundations and Trends in Programming Languages* **2** (2015), no. 1 1–69.

[175] B. Hardekopf, B. Wiedermann, W. R. Cook, and C. Lin, *A formal specification of pointer analysis approximations*, *In submission to Programming Language Design and Implementation (PLDI)* (2009).

[176] B. Steensgaard, *Points-to analysis in almost linear time*, in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[177] M. Sridharan and S. J. Fink, *The complexity of andersen's analysis in practice*, in *Proceedings of the 16th International Symposium on Static Analysis*, 2009.

[178] V. Kahlon, *Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis*, in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 249–259, 2008.

[179] O. Lhoták and K.-C. A. Chung, *Points-to analysis with efficient strong updates*, in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 3–16, 2011.

[180] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, *A few billion lines of code later: Using static analysis to find bugs in the real world*, *Commun. ACM* (2010) 66–75.

[181] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards, *A novel analysis space for pointer analysis and its application for bug finding*, *Sci. Comput. Program.* (2010) 921–942.

[182] S. Biallas, M. C. Olesen, F. Cassez, and R. Huuck, *Ptrtracker: Pragmatic pointer analysis*, in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pp. 69–73, IEEE, 2013.

[183] M. Buss, S. A. Edwards, B. Yao, and D. Waddington, *Pointer analysis for c programs through ast traversal*, .

[184] K. Kelsey, T. Bai, C. Ding, and C. Zhang, *Fast track: A software system for speculative program optimization*, in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, 2009.

[185] A. Gross, *Evaluation of dynamic points-to analysis*, .

[186] M. Mock, M. Das, C. Chambers, and S. J. Eggers, *Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization*, in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pp. 66–72, 2001.

[187] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers, *Improving program slicing with dynamic points-to data*, in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pp. 71–80, 2002.

[188] D. Trabish, T. Kapus, N. Rinetzky, and C. Cadar, *Past-sensitive pointer analysis for symbolic execution*, in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 197–208, 2020.

[189] A. Machiry, *The need for extensible and configurable static taint tracking for c/c++*, 2017. `https://machiry.github.io/blog/2017/05/31/static-taint-tracking`.

[190] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, *libdft: Practical dynamic data flow tracking for commodity systems*, in *Acm Sigplan Notices*, vol. 47, pp. 121–132, ACM, 2012.

[191] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, *Shadowreplica: efficient parallelization of dynamic data flow tracking*, in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 235–246, ACM, 2013.

[192] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, *Taintpipe: Pipelined symbolic taint analysis.*, in *USENIX Security Symposium*, 2015.

[193] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, *Practical taint-based protection using demand emulation*, in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 29–41, ACM, 2006.

[194] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, *Flexitaint: A programmable accelerator for dynamic taint propagation*, in *High Performance Computer Architecture*, 2008.

[195] Y. Xie, A. Chou, and D. Engler, *Archer: Using symbolic, path-sensitive analysis to detect memory access errors*, in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, 2003.

[196] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, *Bugs as deviant behavior: A general approach to inferring errors in systems code*, in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[197] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, *Using model checking to find serious file system errors*, *ACM Transactions on Computer Systems (TOCS)* **24** (2006), no. 4 393–423.

[198] N. Dor, M. Rodeh, and M. Sagiv, *Cssv: Towards a realistic tool for statically detecting all buffer overflows in c*, in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, (New York, NY, USA), pp. 155–167, ACM, 2003.

[199] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, *Buffer overrun detection using linear programming and static analysis*, in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, (New York, NY, USA), pp. 345–354, ACM, 2003.

[200] M. Zitser, R. Lippmann, and T. Leek, *Testing static analysis tools using exploitable buffer overflows from open source code*, in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 97–106, ACM, 2004.

[201] Y. Xie and A. Aiken, *Context-and path-sensitive memory leak detection*, in *ACM SIGSOFT Software Engineering Notes*, ACM, 2005.

[202] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy, *Flow-insensitive static analysis for detecting integer anomalies in programs*, in *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pp. 334–340, ACTA Press, 2007.

[203] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, *Improving integer security for systems with kint.*, in *OSDI*, 2012.

[204] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, *Detecting format string vulnerabilities with type qualifiers*, in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, (Berkeley, CA, USA), USENIX Association, 2001.

[205] S. Kim, R. Y. C. Kim, and Y. B. Park, *Software vulnerability detection methodology combined with static and dynamic analysis*, *Wireless Personal Communications* (2016) 777–793.

[206] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, *Static program analysis as a fuzzing aid*, .

[207] B. Shastry, F. Maggi, F. Yamaguchi, K. Rieck, and J.-P. Seifert, *Static exploration of taint-style vulnerabilities found by fuzzing*, in *11th USENIX Workshop on Offensive Technologies*, USENIX Association, 2017.

[208] C. Csallner, Y. Smaragdakis, and T. Xie, *Dsd-crasher: A hybrid analysis tool for bug finding*, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **17** (2008), no. 2 8.

[209] D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, *Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis*, in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[210] S. Banerjee, D. Devecsery, P. Chen, and S. Narayanasamy, *Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis*, in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[211] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, *Constantine: Automatic side-channel resistance using efficient control and data flow linearization*, in *CCS' 21*, 2021.

[212] T. Palit, J. Firose Moon, F. Monrose, and M. Polychronakis, *Dynpta: Combining static and dynamic analysis for practical selective data protection*, in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.