UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

Security and Usability Issues in Event-driven Applications

Permalink

https://escholarship.org/uc/item/0p0099h4

Author

Bose, Priyanka

Publication Date

2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

University of California
Santa Barbara

Security and Usability Issues in Event-driven Applications

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

Doctor of Philosophy
in
Computer Science

by

Priyanka Bose

Committee in charge:

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

June 2023

The Dissertation of Priyanka Bose is approved.

Professor Yu Feng

Professor Christopher Kruegel, Committee Co-Chair

Professor Giovanni Vigna, Committee Co-Chair

April 2023

Security and Usability Issues in Event-driven Applications

Copyright © 2023

by

Priyanka Bose

To my beloved parents and husband whose endless support and sacrifices always inspired me to pursue my dreams.

Acknowledgements

Where there is a will, there is way

—Albert Einstein

I began my PhD journey with this quote in mind six and a half years ago. Thanks to the endless support of the people around me, their belief in me and my abilities, even in times of doubt, inspired me to keep pushing forward and to never give up on my goals; finally finding ways to achieve it. Therefore, I would like to express my deepest gratitude to all of these individuals who have been instrumental in my success.

I am grateful to my advisors, Giovanni Vigna and Christopher Kruegel, for their encouragement which has always motivated me to tackle challenging and fascinating research problems. I would also like to express my gratitude to Professor Yu Feng, my collaborator, who introduced me to the world of smart contracts. Our numerous in-depth research discussions have helped me sharpen my critical thinking skills, and his guidance and insights have been invaluable in my development as a researcher. I must thank Tim Robinson, the academic coordinator of SecLab, for shielding me from the administrative and financial complexities of grad school, allowing me to focus solely on my research. I am also thankful for the CS department staff who have always treated me with kindness and humility, making my grad school experience an enjoyable one. I owe a great deal to my brilliant lab mates, who have been instrumental in my PhD journey through their insightful discussions, collaborations, and guidance. I am humbled and grateful for the unconditional support and encouragement I have received from this amazing community of mentors, collaborators, and peers. Their influence has made a significant impact on my development as a researcher and as a person.

I am grateful to have an extraordinary group of friends who have consistently turned my darkest days into brighter ones. I want to express my deepest appreciation to Chad,

Tsatsa, Sukrit Da, and Shayonee Di for their unconditional friendship and support, especially during the most challenging phases of my life. I am also thankful to everyone who has been a source of hope and encouragement, whether directly or indirectly, when I needed it the most. Without the love and support of all of you, I could never have come this far in my life.

I am grateful to my family for their constant love and support throughout my journey. I would like to express my sincere gratitude to my beloved parents, Ma and Baba, for always being there for me, especially during my toughest times. Growing up as the only daughter of a middle-class family with traditional values, I recognize that my ambitious and non-traditional pursuits went against societal norms and was not an easy task for my parents. However, they not only stood by my aspirations but also provided endless love and support that gave me the strength to take on the toughest challenges of my life. I am incredibly fortunate to have parents like them, and I cannot thank them enough for everything they have done for me. I am also grateful to my my mother-in-law, and my late father-in-law who always put my dreams and aspirations first. They were a constant source of motivation for me throughout this journey. I am also thankful to my cousins Mejda and Rumpa di, who showed me immense affection and took care of my parents while I was chasing my dreams. I am grateful to my in-laws Bony, Bunty, Trisha Di, and Dadabhai, who were a constant source of joy and laughter. Their presence in my life made me feel like I was never a thousand miles away from my family. Finally, I would like to express my deepest gratitude to my soulmate, best friend, and husband, whose support has been the bedrock of my journey. Through the highs and lows, he has been a constant source of inspiration, motivation, and constructive criticism. From teaching me the basics of an ls command to helping me fuzz the Linux kernel, he has been an incredible mentor, and I could not have achieved any of this without him. Our research discussions have honed my critical thinking skills and helped me become a better researcher. I am grateful

for his selflessness and unwavering dedication to my success, and I feel incredibly blessed to have him as my partner.

Curriculum Vitæ

Priyanka Bose

Education

2016–2023	Ph.D. in Computer Science University of California, Santa Barbara
2016–2022	MS in Computer Science University of California, Santa Barbara
2013–2015	MS in Computer Science & Engineering Indian Institute of Technology
2007–2011	B.Tech. Computer Science & Engineering Institute of Engineering & Management, Kolkata

Publications

1. **Priyanka Bose**, Dipanjan Das, Fabio Gritti, Nicola Ruaro, Christopher Kruegel, Giovanni Vigna, "*Exploiting the Unfair Advantage: Investigating Opportunistic Trading in the NFT Market*". under submission
2. Fabio Gritti, Nicola Ruaro, Robert McLaughlin, **Priyanka Bose**, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna, "*Confusum Contractum: Confused Deputy Vulnerabilities in Ethereum Smart Contracts*". Usenix Security, 2023
3. **Priyanka Bose**, Dipanjan Das, Saastha Vasan, Sebastiano Mariani, Ilya Grishchenko, Andrea Continella, Antonio Bianchi, Christopher Kruegel, Giovanni Vigna, "*Columbus : Android App Testing Through Systematic Callback Exploration*". International Conference on Software Engineering (ICSE), May 2023.
4. Dipanjan Das, **Priyanka Bose**, Nicola Ruaro, Christopher Kruegel, Giovanni Vigna, "*Understanding Security Issues in the NFT Ecosystem*". Conference on Computer and Communications Security (CCS), November 2022.
5. Dipanjan Das, **Priyanka Bose**, Aravind Machiry, Sebastiano Mariani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, "*Hybrid Pruning: Towards Precise Pointer and Taint Analysis*". Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), June 2022.
6. **Priyanka Bose**, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, Giovanni Vigna, "*Sailfish: Vetting Smart Contract State-Inconsistency Bugs in Seconds*".IEEE Symposium on Security and Privacy (IEEE S&P), May 2022.
7. Dongyu Meng, Michele Guerriero, Aravind Machiry, Hojjat Aghakhani, **Priyanka Bose**, Andrea Continella, Christopher Kruegel, Giovanni Vigna, "*Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug*

Symptoms".ACM ASIA Conference on Computer and Communications Security (ASIACCS), January 2021.

8. **Priyanka Bose**, Viet Tung Hoang, Stefano Tessaro, "*Revisiting AES-GCM-SIV: multi-user security, faster key derivation, and better bounds*". Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt), April 2018.
9. **Priyanka Bose**, Dipanjan Das, Chandrasekharan Pandu Rangan, "*Constant Size Ring Signature Without Random Oracle*". Australasian Conference on Information Security and Privacy (ACISP), April 2015.

Abstract

Security and Usability Issues in Event-driven Applications

by

Priyanka Bose

An application is a computer program designed to run on a device. To ease our daily life, we delegate many tedious tasks to these applications. An event-driven application is one where the events drive an application from one state to the other. For example, in the case of Android apps, clicking UI buttons perform certain actions which change the app state. Here, clicking the button is an example of an event. Similarly, for smart contracts, which are very popular nowadays, the execution of a transaction, which can be thought of as an event, drives the state of the smart contract into a different one.

These event-driven applications suffer from both usability and security issues that can be abused by malicious actors. For example, a bug in an Android app may cause the device to become unresponsive or crash altogether. Frequent such crashes result in the instability of the app and a bad user experience. Moreover, app crashes due to a programming error, such as a null pointer exception, may create an opportunity for a malicious user to exploit the vulnerability and execute arbitrary code on the device. For decentralized applications, that use smart contract, a vulnerability in a contract can be exploited by a malicious actor leading to tremendous losses, as demonstrated by recent attacks [1, 2, 3, 4]. For instance, the notorious "TheDAO" [5] reentrancy attack led to a financial loss of about $50M in 2016. Furthermore, in recent years, several other reentrancy attacks, *e.g.*, Uniswap [6], Burgerswap [7], Lendf.me [8], resulted in multimillion dollar losses. Furthermore, given the high popularity and significant total value locked in decentralized applications, they have become attractive targets for various money-making opportunities for malicious

actors. These bad actors may seek to exploit weaknesses in the applications to engage in high-frequency trading activities such as front-running and back-running or to corner the market by buying NFTs (Non-fungible tokens) and selling them later at a significant profit.

Hence, it is crucial to comprehensively analyze and understand the security and usability issues associated with event-driven applications. This is particularly important given the potential financial losses and negative impact on user experience that may result from vulnerabilities in these applications However, these event-driven applications typically have multiple entry points and are highly stateful, allowing anyone to invoke these entry points independently and in any order—making the automated analysis challenging.

Throughout my Ph.D. research, I focused on analyzing various aspects related to the security and usability issues of these event-driven applications and extensively discussed the findings in my dissertation. First, I introduce the fundamental differences between traditional applications and event-driven applications and highlight the unique challenges these event-driven applications pose. Next, I present a comprehensive threat model for these applications with associated security, usability issues, and risks. Lastly, I present in detail how my work focuses on analyzing these applications. Specifically, I present Columbus, a callback-driven Android app testing technique that employs a combination of static analysis, under-constrained symbolic execution and type-guided dynamic heap introspection to generate valid and effective inputs to test the stability and usability of these apps. Furthermore, I developed Sailfish, a scalable system for automatically finding state-inconsistency bugs in smart contracts. Finally, my research delved into the intriguing economic landscape of decentralized applications, with a particular focus on the emerging field of NFT trading—exploring how actors in this ecosystem make use of these unique digital assets to earn profits through high-frequency trading activities,

sometimes in malicious ways.

Contents

Chapter 1

Introduction

An event-driven application is a software program that responds to user inputs or system events by triggering appropriate actions or functions. For an event-driven app, a program can be thought of as divided into different components where each component can serve as an independent entry point to the program. When an external user input or system event occurs, any relevant component can be executed to respond to the event. This differs from traditional program-driven applications, where the execution starts at the beginning of the program and follows a predetermined path, invoking functions or programs as needed before returning the output at the end.

Both Android apps and decentralized apps follow an event-driven paradigm. While Android apps run on a single device, decentralized apps are designed to operate on a distributed network of computers. In an Android app, every interaction with the user interface generates an event that triggers a corresponding callback or event-handler. These components are responsible for handling the event and advancing the app to different states accordingly. Similarly, decentralized apps typically have a web-based user interface on the front-end and a smart contract-based back-end. Smart contracts are programs that run on top of the `EVM` and include multiple public functions that serve as independent

entry points. When a user interacts with the web UI of a decentralized app, a transaction is created, and one of its public functions is executed based on the input provided in the transaction.

1.1 Security and usability issues in event-driven apps

As with the traditional program-driven apps, security and usability is a crucial concern for event-driven apps as well.

As of 2021, Android remains the most widely-used mobile operating system, with a global market share of 75% and approximately 2.8B billion active users worldwide [9]. Android apps serve diverse purposes, ranging from email and banking to gaming and beyond. The official Android app market, Google Play Store, has experienced tremendous growth and currently hosts over 2.9M apps, with over 100K apps added every month [10]. However, vulnerabilities in Android apps can result in app crashes, hindering their usability and making them susceptible to attacks such as denial-of-service and remote code execution, among others, which can be exploited by attackers.

Similarly, decentralized apps, or dApps, are built on blockchain technology, which provides a high degree of security through their decentralized and distributed nature. However, decentralized apps are not immune to security threats, and several risks must be considered. One significant security concern is the potential for malicious actors to exploit vulnerabilities in dApps' smart contracts. Smart contracts have seen widespread adoption, with over 45 million [11] instances covering financial products [12], online gaming [13], real estate, and logistics. Therefore, a vulnerability in a contract can lead to tremendous losses, as demonstrated by recent attacks [1, 2, 3, 4]. For instance, the notorious "TheDAO" [5] reentrancy attack led to a financial loss of about $50M in 2016. Furthermore, in recent years, several other reentrancy attacks, *e.g.*, Uniswap [6],

Burgerswap [7], Lendf.me [8], resulted in multimillion dollar losses. To make things worse, smart contracts are *immutable*—once deployed, the design of the consensus protocol makes it particularly difficult to fix bugs. Since smart contracts are not easily upgradable, auditing the contract's source pre-deployment, and deploying a bug-free contract is even more important than in the case of traditional software.

Additionally, decentralized apps face a threat from malicious trading activities, particularly in the context of cryptocurrency trading. Decentralized exchanges allow users to trade cryptocurrencies such as fungible and non-fungible tokens, as well as Ether, which many users consider to be an investment opportunity. The simplest approach is to buy tokens in Ether at one price and sell them later at a higher price, generating a profit equal to the difference in value. However, some savvy traders employ both illicit and benign strategies to maximize their profits. These may include tactics such as frontrunning, backrunning, and arbitraging, among others.

In order to encourage usage of these event-driven apps, it is important to address security and usability issues that may act as deterrents for users. This requires careful analysis and implementation of measures to mitigate these issues and ensure that the apps are both secure and user-friendly. While extensive research has been conducted in this field to study and detect such issues, every technique seems to have its own limitations.

Android apps. Android apps need to be thoroughly tested before developers push them to the market in order to provide a smooth user experience. Modern Android apps use rich user interfaces (UIs) and complex app logic, thus making automated exploration challenging. Android apps are event-driven programs, *i.e.*, each interaction with the UI of the app generates an event, which drives the app through different states. Therefore, synthesizing a correct sequence of events is essential to efficiently explore the state space of an app. Many prior techniques rely on UI testing frameworks [14, 15, 16, 17, 18, 19, 20] to exercise the app by generating appropriate events. However, a large class of events

is widget-specific, and requires multiple user actions to be taken in a specific order at specific UI coordinates. Given the variety of the Android widgets, and the different types of events they support, this is non-trivial. To address this, callback-driven approaches [21] leverage the fact that when a UI event is triggered, the associated *event handler*, also known as *callback*, is executed. Callbacks are the methods in the app typically invoked by the Android framework on the occurrence of an event, *e.g.*, `click` on a widget. Callback-driven techniques call those callbacks directly—essentially eliminating the need for event generation altogether.

Existing callback-driven approaches suffer from two main limitations. **(L1)** They assume the knowledge of both the Android callbacks and the APIs to determine *what* to call and *how*, respectively. Given an app, the first challenge is to identify its callbacks. For that, existing tools maintain a fixed and often small list of supported callbacks. Once a callback is identified, it has to be invoked with arguments that match the types that the callback expects. Callbacks accept two types of arguments: *primitive*, *e.g.*, `int`, and `float`, or *objects*. Object arguments are harder to deal with. Prior techniques depend on a human expert for writing the necessary *driver* code, which would leverage widget-specific Android APIs to retrieve live objects from the app context, so that those can be supplied as arguments. Since adding support for a callback requires a non-trivial manual effort, it is hard to extend the support for all the callbacks in the framework. Quite understandably, while there are approximately 19,647 callbacks in Android 4.2 [22], the state-of-the-art callback-driven testing tool EHBDroid [21] supports only 58 of them. **(L2)** Apps accept user-supplied data as input, *e.g.*, text. Only generating event sequences, which existing tools focus on, is not enough, because certain functionalities may only be reachable under specific input. For example, a payroll app calculates tax differently depending on the income of an employee. To address these challenges, an automated app testing technique must be able to identify callbacks and supply appropriate input for testing.

Smart contracts. Smart contract vulnerabilities can result in huge financial losses. One such vulnerability class is *state-inconsistency* (SI) bugs that enable an attacker to manipulate the global state, *i.e.*, the storage variables of a contract, by tampering with either the order of execution of multiple transactions (*transaction order dependence* (TOD)), or the control-flow inside a single transaction (*reentrancy*). In those attacks, an attacker can tamper with the critical storage variables that transitively have an influence on money transactions through data or control dependency. Though "TheDAO" [5] is the most well-known attack of this kind, through an offline analysis [23, 24] of the historical on-chain data, researchers have uncovered several instances of past attacks that leveraged state-inconsistency vulnerabilities.

While there are existing tools for detecting vulnerabilities due to state-inconsistency bugs, they either aggressively over-approximate the execution of a smart contract, and report false alarms [25, 26], or they precisely enumerate [27, 28] concrete or symbolic traces of the entire smart contract, and hence, cannot scale to large contracts with many paths. Dynamic tools [24, 23] scale well, but can detect a state-inconsistency bug only when the evidence of an active attack is present. Moreover, existing tools adopt a syntax-directed pattern matching that may miss bugs due to incomplete support for potential attack patterns [25].

A static analyzer for state-inconsistency bugs is crucial for the pre-deployment auditing of smart contracts, but designing such a tool comes with its unique set of challenges. For example, a smart contract exposes public methods as interfaces to interact with the outside world. Each of these methods is an entry point to the contract code, and can potentially alter the persistent state of the contract by writing to the storage variables. An attacker can invoke *any* method(s), *any* number of times, in *any* arbitrary order—each invocation potentially impacting the overall contract state. Since different contracts can communicate with each other through public methods, it is even harder to detect a cross-function attack

where the attacker can stitch calls to multiple public methods to launch an attack. Though Sereum [24] and Ecfchecker [29] detect cross-function attacks, they are dynamic tools that reason about one single execution. However, statically detecting state-inconsistency bugs boils down to reasoning about the entire contract control and data flows, over multiple executions. This presents significant scalability challenges, as mentioned in prior work [24].

Cryptocurrency trading. 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* [30]. The authors in FlashBoys [31] 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.* [32] quantified the probability of being able to perform such an attack and the profits it can yield. In fact, a recent paper [33] reported the profit extracted from the blockchain to be a staggering $28.8M USD in just two years, leveraging sandwiching, liquidation, and arbitrage. 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.* [34] analyzed how flashloans have been used to execute arbitrage and oracle manipulation attacks. DefiPoser [35] proposes trading algorithms to generate profit by crafting complex DeFi transactions, both with and without flashloans.

Previous research has mostly focused on market manipulation strategies concerning fungible tokens and Ether. However, non-fungible tokens (NFTs) have also gained significant value and several NFT marketplaces (NFTMs) have emerged in recent years to facilitate their buying and selling, including well-known platforms such as OpenSea,

Rarible, and Axie. These NFTMs have seen tremendous growth, with OpenSea alone generating 236M USD in platform fees and facilitating a trading volume of 3.5B USD in August 2021[36]. Moreover, individual NFT sales have seen an unprecedented surge in value[37], with nine out of the top ten most expensive sales [38] occurring between February and August 2021. As a result, malicious actors have entered the NFT space to employ adversarial trading strategies to make quick profits.

However, analyzing such trading activities surrounding NFTs is challenging due to several factors. Unlike ERC-20 exchanges, NFTM are order-book based. Additionally, since each NFT is unique, traders need to specifically buy the token they want, and the marketplace will not automatically fill orders. Moreover, the value of NFTs is not standardized and is entirely based on perceived value, making their valuation tricky. Also, NFT trading strategies can vary significantly based on different NFT actions. Therefore, the analysis used for ERC-20 exchanges is not applicable to the NFT ecosystem, and a different approach is necessary to analyze NFT trading.

1.2 Contributions.

This dissertation details my research on the investigation and analysis of security and usability issues in event-driven applications. I have examined two such real-world event-driven apps and conducted an automatic analysis of their security, usability issues, and associated risks. This investigation revealed that different types of event-driven apps present distinct challenges, which are discussed in detail in this dissertation. Additionally, I demonstrate that using a combination of program analysis techniques enables the detection of such issues in event-driven apps in a more scalable and precise manner. In summary, this dissertation presents the following contributions:

- We propose a callback-driven Android app testing approach by presenting **(i)**

a generic technique to extract all the callbacks present in an app, and **(ii)** an analysis based on under-constrained symbolic execution (primitive arguments), and *type-guided* dynamic object filtering for generating valid arguments to invoke callbacks. Further, we make the app exploration systematic by integrating two novel feedback mechanisms: **(i)** a *data dependency* feedback that increases the probability of triggering bugs due to uninitialized variables, and **(ii)** a *crash-guided* dynamic scoring mechanism that prevents us from rediscovering the same bugs. We implement the proposed technique in a practical tool called Columbus, and we make it publicly available [39]. Our evaluation demonstrates that Columbus outperforms the state-of-the-art tools both in terms of code coverage and the number of unique crashes that it identifies.

- We define state-inconsistency vulnerabilities and identify two of its root-causes, including a new reentrancy attack pattern that has not been investigated in the previous literature. We model state-inconsistency detection as hazardous access queries over a unified, compact graph representation (called a *storage dependency graph* (SDG)), which encodes the high-level semantics of smart contracts over global states. We propose a novel *value-summary analysis* that efficiently computes global constraints over storage variables, which when combined with symbolic evaluation, enables Sailfish to significantly reduce false alarms. We perform a systematic evaluation of Sailfish on the entire data set from Etherscan. Not only does Sailfish outperform state-of-the-art smart contract analyzers in terms of both run-time and precision, but also is able to uncover 47 zero-day vulnerabilities (out of 195 contracts that we could manually analyze) not detected by any other tool.

- We study high-frequency opportunistic trading of NFTs. We first present a systematic overview of how NFT trading works, and what are the different trade

actions performed by the buyers and sellers. We then study previous instances of opportunistic trades, and organized them according to the strategies employed by the traders, *viz.*, acquire, instant profit generation, and loss minimization. Next, we analyze strategies used to acquire high-value NFTs for long-term holding. NFTs acquired this way are meant to be held for long, and sold when the market is assessed to be favorable. We analyze strategies used to make an instant profit in the NFT market. NFTs acquired this way are meant to be held for a short time, and are typically sold in the same transaction, thus making an instant profit. Lastly, we analyze strategies used to minimize potential losses incurred from an NFT purchase.

The remainder of this dissertation is structured as follows:

In Chapter 2, we introduce COLUMBUS, a callback-driven testing technique that employs two strategies to eliminate the need for human involvement: **(i)** it automatically identifies callbacks by simultaneously analyzing both the Android framework and the app under test; **(ii)** it uses a combination of under-constrained symbolic execution (*primitive* arguments), and *type-guided* dynamic heap introspection (*object* arguments) to generate valid and effective inputs. Lastly, COLUMBUS integrates two novel feedback mechanisms—*data dependency* and *crash-guidance*—during testing to increase the likelihood of triggering crashes and maximizing coverage. In our evaluation,COLUMBUS outperforms state-of-the-art model-driven, checkpoint-based, and callback-driven testing tools both in terms of crashes and coverage.

In Chapter 3, we present SAILFISH, a scalable system for automatically finding state-inconsistency bugs in smart contracts. To make the analysis tractable, we introduce a hybrid approach that includes (i) a light-weight exploration phase that dramatically reduces the number of instructions to analyze, and (ii) a precise refinement phase based on symbolic evaluation guided by our novel value-summary analysis, which generates

extra constraints to over-approximate the side effects of whole-program execution, thereby ensuring the precision of the symbolic evaluation. We developed a prototype of SAILFISH and evaluated its ability to detect two state-inconsistency flaws, *viz.*, reentrancy and transaction order dependence (TOD) in Ethereum smart contracts. Our experiments demonstrate the efficiency of our hybrid approach as well as the benefit of the value summary analysis. In particular, we show that SAILFISH outperforms five state-of-the-art smart contract analyzers (SECURIFY, MYTHRIL, OYENTE, SEREUM and VANDAL) in terms of performance, and precision. In total, SAILFISH discovered 47 previously unknown vulnerable smart contracts out of 89,853 smart contracts from ETHERSCAN.

In Chapter 4, we analyze and study the high-frequency opportunistic trading of NFTs in decentralized exchanges. Through this study, we have identified several illicit and profit-making trading strategies that are employed by malicious actors to earn profits. In addition, we show that not all NFT trading strategies result in profit; some can result in substantial losses for traders..

Finally, Chapter 5 contains the conclusions drawn from my work and future directions.

Chapter 2

Columbus: Android App Testing Through Systematic Callback Exploration

In this chapter, we present COLUMBUS, a callback-driven Android app testing technique. Android apps are event-driven programs, *i.e.*, each interaction with the UI of the app generates an event, which drives the app through different states. Therefore, synthesizing a correct sequence of events is essential to efficiently explore the state space of an app. Many prior techniques rely on UI testing frameworks [14, 15, 16, 17, 18, 19, 20] to exercise the app by generating appropriate events. However, a large class of events is widget-specific, and requires multiple user actions to be taken in a specific order at specific UI coordinates. As we explain in Section 2.2, the `onDateChanged` event of the `DatePickerDialog` widget is one such example. Generating such events *deterministically* is challenging for a UI-based testing tool, unless it has been equipped with the knowledge of how to generate all the correct events. Given the variety of the Android widgets, and the different types of events they support, this is non-trivial. To address this, callback-driven approaches [21] leverage the

fact that when a UI event is triggered, the associated *event handler*, also known as *callback*, is executed. Callbacks are the methods in the app typically invoked by the Android framework on the occurrence of an event, *e.g.*, `click` on a widget. Callback-driven techniques call those callbacks directly—essentially eliminating the need for event generation altogether.

Existing callback-driven approaches suffer from two main limitations. **(L1)** They assume the knowledge of both the Android callbacks and the APIs to determine *what* to call and *how*, respectively. Given an app, the first challenge is to identify its callbacks. For that, existing tools maintain a fixed and often small list of supported callbacks. Once a callback is identified, it has to be invoked with arguments that match the types that the callback expects. Callbacks accept two types of arguments: *primitive*, *e.g.*, `int`, and `float`, or *objects*. Object arguments are harder to deal with. Prior techniques depend on a human expert for writing the necessary *driver* code, which would leverage widget-specific Android APIs to retrieve live objects from the app context, so that those can be supplied as arguments. Since adding support for a callback requires a non-trivial manual effort, it is hard to extend the support for all the callbacks in the framework. Quite understandably, while there are approximately $19,647$ callbacks in Android 4.2 [22], the state-of-the-art callback-driven testing tool EHBDROID [21] supports only 58 of them. **(L2)** Apps accept user-supplied data as input, *e.g.*, text. Only generating event sequences, which existing tools focus on, is not enough, because certain functionalities may only be reachable under specific input. For example, a payroll app calculates tax differently depending on the income of an employee.

We present COLUMBUS, an Android app testing technique that addresses both the challenges. To address **L1**, COLUMBUS adopts a two-phase approach. First, we statically identify all the callbacks present in the app (*what* to call). Specifically, our *callback discovery* module statically extracts all the callback signatures $\mathcal{L}$ supported by the Android framework. Since an app has to override a framework callback to provide its own imple-

mentation, we use $\mathcal{L}$ to identify the callback implementations present in the app. Once callbacks are identified, then we dynamically prepare arguments (*how* to call) to invoke them with. Unlike previous techniques that rely on manually-written, callback-specific driver code to generate object arguments, we resort to a hybrid approach. Our *exploration* module performs a dynamic introspection of the app's heap at run-time, followed by a *type-guided* object filtering to supply appropriate arguments to the callback. This callback discovery and argument generation strategies together insulate COLUMBUS from the complexity of the Android API and obviate the need for any prior knowledge. To address **L2**, we leverage the fact that many user inputs are of primitive types, and often appear as the arguments to the callbacks. Therefore, the *argument generation* module symbolizes the primitive arguments of a callback, and performs an under-constrained symbolic execution to generate the possible values of those arguments to drive the execution along all paths. Symbolic execution is scoped within a single callback instead of the entire app to maintain a balance between precision and scalability.

In addition to tackling those two limitations, we integrate two novel feedback mechanisms into our exploration loop. **(i)** The *callback dependency* module passes on statically-identified data-dependencies between callbacks as feedback, which enables COLUMBUS to generate callback sequences that increase the likelihood of triggering crashes due to uninitialized objects, *e.g.*, `NullPointerException`. **(ii)** We design a *crash-guided* dynamic scoring mechanism that gradually deprioritizes crash-inducing paths in the app to drive the exploration towards unexplored code. In effect, COLUMBUS is incentivized to discover more crashes than rediscovering the already found ones.

We evaluated COLUMBUS on 60 apps of the AndroTest [40] benchmark, and top 140 real-world apps from the Google Play Store. Compared to the state-of-the-art model-based techniques STOAT [17] and APE [20], checkpoint-based technique TIMEMACHINE [41], and callback-driven technique EHBDROID [21], COLUMBUS achieves 12%, 5%, 6%, and

31% more in average coverage, and discovers 4.42, 1.23, 1.57, and 5.48 times more crashes on the AndroTest apps, respectively. COLUMBUS is also able to find 70 crashes in 54 real-world apps.

2.1 Background

Android events. Android apps are event-driven programs. That is, apps behave as state machines, and events cause a transition from one state to the other. An event is generated in response to one or more user actions (UI events), or by Android itself (system events). Examples of UI events include `click`, `drag`, `pan`, `pinch`, `zoom`, *etc.* Modern Android devices are equipped with peripherals, such as, Bluetooth and WiFi, and sensors like motion sensors and accelerometers. Any change in the state of these devices is detected by the OS, which then generates a system event to notify "interested" apps. Examples of system events are Bluetooth disconnected, phone tilted, and low battery level.

Based on the number of actions needed to generate an event, we define two types of events: *primitive* and *composite*. Primitive events are either system events or UI events generated due to a single action. For example, `MotionEvent` (`ME`) reports the movement of an input device like a mouse, pen, finger, trackball, or `KeyEvent` reports key and button related actions. A composite event consists of multiple primitive ones, which are sequenced with strict spatial and temporal requirements. Say, we want to drag an object from point p_1, and drop it at point p_n along the trajectory $[p_1, p_2, p_3, ..., p_n]$. In order to programmatically generate a `drag` event, the following sequence (temporal) of primitive events need to be fired at those exact coordinates (spatial): `ME.ACTION_DOWN` $(p_1) \rightarrow$ $\{$`ME.ACTION_MOVE` $(p_i) \mid 2 \leq i \leq (n-1)\} \rightarrow$ `ME.ACTION_UP` (p_n). Without the support for a composite event, it is nearly impossible for a UI testing tool to generate most of them just 'by chance'. To make matter worse, numerous such composite events are

widget-specific, *e.g.*, the `DateChanged` event recognized by `DatePickerDialog`. Therefore, adding support for individual events in a UI testing tool is nearly impossible.

Android callbacks. An Android *callback*, also known as an *event handler*, is a piece of code that the framework invokes when a specific event takes place, for example; the `onClick` callback is called when a `click` event occurs. Typically, the framework only provides empty callbacks, which an app selectively overrides to respond to the respective events. When an event is generated, it is broken down into `Messages`, which are then put into a `MessageQueue` managed by the `Looper`, the entity that runs the message loop. The `Looper` processes the `Messages` in first-in-first-out order, and calls the associated callbacks. While invoking a callback, the framework supplies the appropriate arguments, which can be of two types—*primitive*, *e.g.*, `int`, `float`, *etc.*, or `object`, *i.e.*, an instance of a class.

Android activity: An activity is a UI element that acts as a container of other UI elements. It often presents itself in the form of a window. Activities are managed by maintaining an activity stack. When a new activity starts, it is placed on the top of the stack, while the previous one is paused, and remains below the current one in the stack. A paused activity does not come to the foreground again until the current activity exits. An activity transitions through different states of its *lifecycle* as a user navigates through an app. Lifecycle callbacks, *e.g.*, `onCreate`, `onPause`, `onResume`, are the ones associated with such lifecycle events.

2.2 Motivation and challenges

This section introduces a motivating example, the challenges it presents to the state-of-the-art callback-driven app testing tools, and how we tackle them.

The code in Figure 2.1 shows three callbacks that an Android app might implement. The callback functions are executed when the user interacts with specific UI elements, *i.e.*, clicks

```
1  protected void onListItemClick(ListView l, View v, int position, long id) {
2    File f = (File)(mList.get(id).get(ITEM_KEY_FILE));
3    if (f.isFile()) {
4      mSelectedFile = f;
5      showDialog(DIALOG_IMPORT_FILE);
6    }
7  }
8
9  public void onClick(DialogInterface dialog, int whichButton) {
10   File f = mSelectedFile;
11   Intent i = new Intent(mContext, myActivity.class);
12   Uri u = Uri.fromFile(f);
13   i.setData(u);
14   startActivity(i);
15 }
16
17 public void onDateChanged(DatePicker view, int year, int month, int day) {
18   if (day == 15 && month == 6 && year == 2020)
19     Toast.makeText(context, "Success!", ...).show();
20 }
```

Figure 2.1: Code containing three callbacks. Their data dependencies (■) and checks on the arguments (■) are highlighted.

on a list item, clicks on a button, and sets a date using a `DatePickerDialog` (Figure 2.2), respectively. UI-based testing tools [14] generate events, *e.g.*, clicks, to interact with the UI of such apps. However, these tools are not widget-aware, meaning that, they are unable to *systematically* generate composite events unless they already know how to generate them. For example, the following events need to be generated in an exact sequence, on specific UI elements, to call the `onDateChanged` callback—**(i)** `DatePickerDialog` widget is clicked to bring up the spinner control, **(ii)** the day/month/year is changed by clicking on the up/down arrows, and **(iii)** the `Set` button is clicked. It is unlikely for a UI-based testing tool to be able to deterministically generate this event sequence without any guidance. Moreover, to set a particular date, the up/down arrows need to be clicked a specific number of times—which is hard as well. To overcome this limitation, callback-driven techniques [21] invokes the callback, *e.g.*, `onDateChanged`, directly bypassing the UI layer altogether. While callback-driven testing shows promise, it still suffers from the following

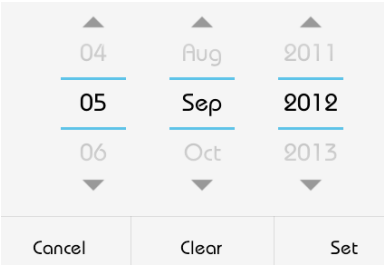

Figure 2.2: A `DatePickerDialog` widget

```
1  void onCreate(Bundle bundle) {
2    ListView lv = getListView();
3  }
4
5  void ehbTest() {
6    for (int i=0; i<lv.size(); i++) {
7      View v = lv.getChildAt(i);
8      long id = lv.getAdapter()
9                  .getItemId(i);
10     this.onListItemClick(lv,v,i,id);
11   }
12 }
```

Figure 2.3: EHBDROID instrumentation for `onListItemClick()`

limitations.

Identifying callbacks. The first step of callback-driven testing is identifying the callbacks. Unfortunately, the set of callbacks supported by the Android framework is huge. While previous research [22] identified approximately 19,647 callbacks in Android 4.2; EHBDROID, the state-of-the-art callback-driven testing tool, supports only 58 callbacks. COLUMBUS statically analyzes the app and the Android framework together to address this issue (Section 2.3.1).

Providing callback arguments. Callbacks accept either primitive arguments or objects. The primitive arguments are often involved in path conditions within the callback. Without the correct value of such primitives, part of the callback may never be exercised. In Figure 2.1, the `Toast` message appears only on a specific date. Existing callback-based testing tools use a set of predefined values to invoke callbacks. Therefore, Line 19 will possibly never be explored. COLUMBUS symbolizes primitive arguments and employs under-constrained symbolic execution to infer values to make larger part of the callback code reachable (Section 2.3.2).

For object arguments, such as, the `ListView` and `View` arguments of the `onListItemClick` callback in Figure 2.1, callback-driven tools use the Android API (by statically

instrumenting the app) to retrieve correct objects from the app context, as shown in Figure 2.3 (Line 2 and Line 7). However, this approach is not scalable, as the number of callbacks in the Android framework is huge, and the tool requires adding explicit support for all the arguments of all the callbacks. Instead, COLUMBUS retrieves live objects from the app heap at runtime, and then applies *type-guided* object filtering to provide the correct arguments (Section 2.3.2). Type information comes from a one-time, static, pre-processing phase.

Data dependency feedback. Variables are often shared among multiple callbacks. Shared data introduces data dependencies, which an app should either enforce by restricting available UI actions, or handle by placing a sanity check. In Figure 2.1, both the `onClick` and `onListItemClick` callbacks use the same variable `mSelectedFile`. Specifically, `onListItemClick` opens a file, and sets the file handle `mSelectedFile` (Line 4), which `onClick` uses in Line 10. This implies that `onListItemClick` has to be invoked before `onClick`, otherwise the `onClick` method would generate a `NullPointerException`. COLUMBUS statically infers such data dependencies and passes the same as feedback during testing. While synthesizing a callback sequence, COLUMBUS attempts to violate the expected order to increase the likelihood of inducing crashes (Section 2.3.3).

2.3 The Columbus framework

In this work, we propose COLUMBUS, a framework to test Android apps by directly invoking their callbacks. For a given Android app, COLUMBUS first identifies its callbacks (Section 2.3.1). It then obtains the primitive argument values that correspond to different execution paths in these callbacks (Section 2.3.2) and identifies inter-callback dependencies (Section 2.3.3). Finally, our tool invokes the identified callbacks—**(i)** in orders that initially violate (to increase the chances of triggering uninitialized data-related bugs), and later

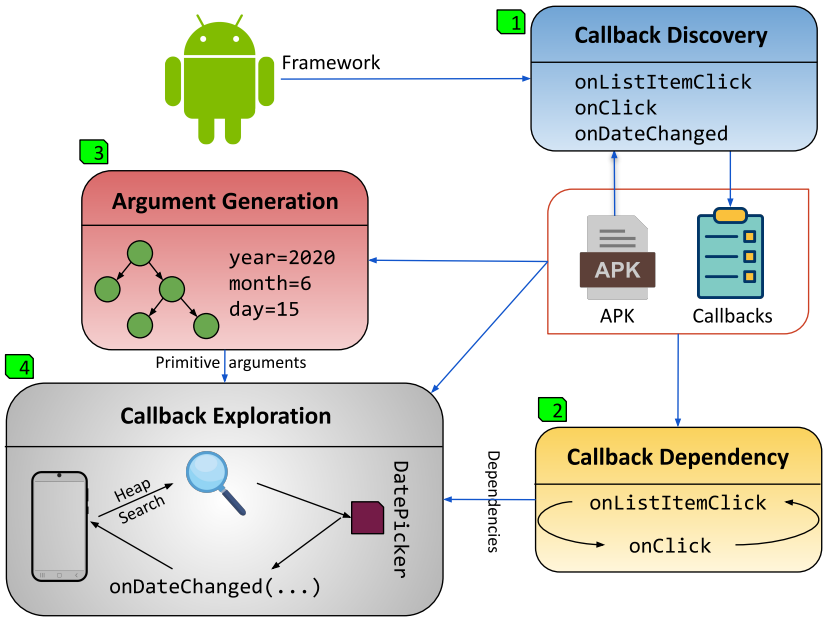

Figure 2.4: Overview of COLUMBUS with reference to the motivating example in Figure 2.1

respect their dependencies, **(ii)** with their expected arguments during the exploration (Section 2.3.4). COLUMBUS keeps track of the callback-defining classes explored during the app execution, and gives higher priority to exploring classes that have been less explored. Figure 2.4 depicts the high-level workflow of our system.

2.3.1 Callback discovery

Every Android app defines its own set of callbacks. Though state-of-the-art approaches [21] resorted to a predefined set of callbacks, the Android framework contains thousands [22] of callbacks, and the number is constantly increasing. In order to facilitate effective app exploration, in this work, we present an approach to automated callback discovery. COLUMBUS's callback identification is presented in Algorithm 1. At a high level, our callback discovery approach first statically analyzes the framework (Function `AndroidFrameworkAnalysis`) followed by an analysis of the app under test (Function `AppAnalysis`), and outputs a list of callbacks present in the app.

Algorithm 1: Static callback identification

```
1  Function AndroidFrameworkAnalysis
       Input  : Android framework JAR
       Output : Classes with callback candidates Δ
2      Δ ← {}
3      CG_f ← GetCallGraph(JAR)
4      CH_f ← GetClassHierarchy(JAR)
5      foreach class c_f ∈ GetClassesFromJar(JAR) do
6          M_f ← ∅
7          foreach method m_f ∈ GetMethodsFromClass(c_f) do
8              if IsPublicOrProtected(m_f) then
9                  if GetCallers(c_f, m_f, CG_f) ≠ ∅ then
10                     M_f ← M_f ∪ m_f
11                 end
12             end
13         end
14         Δ[c_f] ← Δ[c_f] ∪ M_f
15     end
16     foreach (c_f, M_f) ∈ Δ do
17         foreach subclass c'_f ∈ GetSubClasses(c_f) do
18             M'_f ← Δ[c'_f]; M'_f ← M'_f ∪ M_f; Δ[c'_f] ← M'_f
19         end
20     end
21     return Δ, CH_f
22 Function AppAnalysis
       Input  : App's APK, Framework classes with callback candidates Δ, Framework's class
                hierarchy CH_f
       Output : Application callbacks CB
23     CB ← ∅
24     foreach class c_a ∈ GetClassesFromApk(APK) do
25         ClassAndItsParents ← c_a ∪ GetSuperClasses(c_a)
26         foreach cp_a ∈ ClassAndItsParents do
27             foreach (c_f, M_f) ∈ Δ do
28                 if cp_a extends c_f ∨ cp_a implements c_f then
29                     foreach m_a ∈ GetClassMethods(cp_a) do
30                         foreach m_f ∈ M_f do
31                             if IsCompatible(m_f, m_a) then
32                                 CB ← CB ∪ m_a
33                             end
34                         end
35                     end
36                 end
37             end
38         end
39     end
40     return CB
```

Android framework analysis. Our analysis is based on two observations. **(i)** As discussed in Section 4.1, in order to perform the intended action once an event is generated, an app needs to override the respective callback present in the Android framework. To be overridden, a callback needs to be declared as either a *protected*, or a *public* method within the framework. **(ii)** Moreover, at runtime, callbacks are typically invoked within the framework through a series of internal method calls once an event is generated—meaning that, callbacks have caller(s) within the framework.

COLUMBUS first constructs the framework's callgraph CG_f. To build the call graph, COLUMBUS performs intra-procedural type inference [42] to determine the possible dynamic types of the object on which a method is called. When this fails, COLUMBUS then over-approximates the possible targets as all the subclasses of its static type. Now, for every method m_f in a framework class c_f, COLUMBUS considers m_f as a potential callback (Lines $7--13$) if—**(i)** m_f is declared as either *protected*, or *public*, and **(ii)** m_f has at least one caller in CG_f. At the end, we compute a mapping Δ that maps each class c_f to their potential callbacks. Each callback m_f is a tuple, which consists of the defining class c_f, the method name, and the types of its arguments. Now, this mapping Δ is incomplete, because a class can inherit callbacks from its superclasses as well. Therefore, COLUMBUS computes the complete list of potential callbacks for every c_f by walking up the class hierarchy to consolidate superclass callbacks, too (Lines $16--20$). The updated callback mapping Δ and the class hierarchy information CH_f are returned as the output. Note that COLUMBUS performs the framework analysis once per framework.

The above analysis is inspired by EdgeMiner [22]. The main goal of EdgeMiner is to detect framework callbacks, and using that to discover the registration methods within the framework. However, the end goal of Columbus is to detect application level callbacks by leveraging the framework callbacks.

Android app analysis. The goal of this phase is to find whether any app class method

m_a is a valid overriding method of the framework class callback m_f. In order to override a callback within an app, the app class c_a needs to either extend or implement the corresponding callback-defining class c_f of the Android framework. For example, in Figure 2.1, to override the `onListItemClick` callback, the app class needs to extend the `ListActivity` framework class. COLUMBUS identifies such pairs of classes (c_f, c_a) by statically analyzing the app. In the next step, it checks whether any app method $m_a \in c_a$ has the same name and the same number of arguments as any framework method $m_f \in c_f$, and the arguments of m_a are *type-compatible* with those of m_f (Lines $29 - -35$). We call a type t_1 to be *compatible* with another type t_2, if either $t_1 = t_2$, or t_1 is a subclass of t_2 according to the class hierarchy. To determine type compatibility, COLUMBUS constructs the full class hierarchy by unifying ($\oplus$) the framework class hierarchy CH_f with the app class hierarchy CH_a. Let $A \rightarrow B$ denote that A is a superclass of B. Now, if the relations $H_1 = A \rightarrow B$ and $H_2 = B \rightarrow C$ appear in CH_a and CH_f, respectively, then $H_1 \oplus H_2 = A \rightarrow B \rightarrow C$. Finally, we obtain the set of potential callbacks in an app. Our analysis would discover all three functions `onListItemClick`, `onClick`, and `onDateChanged` in Figure 2.1 as callbacks.

Identifying callbacks by analyzing either the app, or the framework alone is challenging. Since a callback is invoked by the framework, the callback methods do not have incoming edges visible from the call graph of the app. However, an analysis relying only on this fact alone will generate false positives—because, it could detect a non-callback method as a callback due to the inherent incompleteness of Java call graphs [43]. Similarly, our framework analysis is over-approximated in a way that will definitely contain the callbacks, but non-callbacks methods, too. Intuitively, therefore we 'intersect' the framework callback candidates and app methods to determine the true callbacks.

During this phase, we can encounter methods of a generic Android framework class `Object`, that are declared as `public`, and can therefore be overridden by the corresponding

application-level classes inheriting the `Object` class. The number of such callbacks appearing as part of the final callback list was negligible (around 3%). We do not consider such methods as callbacks.

2.3.2 Generating arguments for callbacks

In order to invoke a callback, we need to provide argument values conforming to the correct types. In case of GUI-action-driven exploration strategies, the framework provides these arguments, which are derived from the events resulting from the GUI actions. Therefore, to invoke callbacks *without* relying on GUI actions, COLUMBUS needs to tackle the challenge of generating arguments for these callbacks, with a goal to explore the paths within a callback resulting in faster coverage and better crash discovery.

A callback argument can be one of two types: primitive or reference. For each type, COLUMBUS uses different strategies to generate the corresponding arguments.

Primitive type arguments.

Primitive type arguments, *e.g.*, `integer`, `long`, `string`, and `boolean`, are typically involved in program paths that can only be explored with a specific set of values. For instance, Line 19 of the `onDateChange` callback in Figure 2.1 will get executed *only if* the integer arguments `day`, `month`, and `year` are equal to 15, 6, and 2020. Therefore, to effectively explore all the paths in such a callback without resorting to a computationally expensive random search, COLUMBUS needs to provide these specific set of values to the callback during invocation. In this case, COLUMBUS symbolizes respective callback arguments, and performs an under-constrained symbolic execution (until termination, or time-out) to generate concrete values.

Precisely, COLUMBUS starts the symbolic execution at the entry point of each of

the callbacks, and collects constraints on the arguments corresponding to each of the execution paths. It then solves these constraints and generates concrete argument values, which when provided as arguments to the callback during invocation, result in exercising those paths within the callback. During symbolic execution, we track constraints on objects that modify the program state, such as **(i)** callback arguments, and **(ii)** API return values.

Callback arguments. COLUMBUS executes the callback with symbolic and unconstrained arguments. It then collects the constraints in each of the execution paths that involve operations on the symbolic arguments. For example, if one of the arguments is an object, and during execution, one of its fields is set to 5, COLUMBUS's symbolic execution engine will automatically add a constraint stating that the specific attribute needs to be equal to 5 (to follow a particular program path of interest).

API calls. COLUMBUS's symbolic execution engine generates summaries for common functions, for example, the Java runtime function `exit()`. These summaries capture the side effects of these APIs that modify the program state. For APIs without a summary, we return a fresh symbolic value conforming to the return type of the API.

COLUMBUS's symbolic execution engine is capable of generating concrete values of `integer`, `float`, `boolean`, and constant `string` types.

Reference type arguments.

Reference type argument objects frequently represent UI elements where a user performs certain actions. In Figure 2.1, when a user clicks on `AlertDialog` (a subclass object of `DialogInterface`), the framework invokes the `onClick` callback with an argument object of type `AlertDialog`. Therefore, to invoke the `onClick` callback without relying on the Android framework, we need to provide an object of type `DialogInterface`, or a subclass of `DialogInterface`—as an argument.

App heap search. During the app exploration (Section 2.3.4), as and when new `Activities` are visited, these object instances are created in the app heap. Therefore, in order to invoke a callback that requires reference type arguments, COLUMBUS monitors the app heap by dynamically instrumenting the app under test. In many cases, the argument type present in the callback signature is not the one created in the app heap. In Figure 2.1, the `onClick` callback has an argument of type `DialogInterface`. However, the object created will be of type `AlertDialog`, a subclass of `DialogInterface`. To account for this scenario, *i.e.*, if an object instance of a reference type inferred from the callback signature is not available in the app heap, COLUMBUS searches for object instance(s) that is a subclass of the required type.

Custom object creation. It may still happen that no object instances of the required type or its subclass are found in the heap. For example, certain types of objects required as a callback argument, *e.g.*, `KeyEvent`, and `MotionEvent`, that are created by the Android framework *only* when it registers touch, or key-press on UI elements. Therefore, in order to invoke such callbacks, COLUMBUS leverages Java reflection. Specifically, for such a reference, COLUMBUS creates the object using its public constructor. If the constructor expects primitive type arguments, COLUMBUS uses either a random value, or a value from a pre-defined set as the argument. For example, to create `KeyEvent`, or `MotionEvent` objects, COLUMBUS uses pre-defined values as they should be valid screen coordinates in order to successfully explore the callback. If a constructor expects reference type objects, COLUMBUS either finds these objects through app heap search, or creates recursively through Java reflection. For example, if we were to create an object of type `A` which has a constructor that accepts an object of type `B`, then we create objects bottom up (*i.e.*, first `B`, then `A`). In case multiple such constructors exist, COLUMBUS picks the one which requires the least number of reference type arguments.

2.3.3 Inter-callback dependency

Callbacks within an app can share variables resulting in *read-write* data dependencies. As discussed in Section 2.2, for `onListItemClick` and `onClick` callbacks (Figure 2.1), prioritizing dependency-violating order, *i.e.*, invoking `onClick` before `onListItemClick`, brings us faster to a crash discovery. Whereas invoking the callbacks in the dependency-respecting order allows for a better code coverage. For example, the execution of the Lines $13 - -14$ in `onClick` happens only if the reference `mSelectedFile` accessed at Line 10 is defined by a prior execution of `onListItemClick`.

Based on this observation, COLUMBUS computes callback pairs having shared variable dependencies by performing a field-insensitive analysis of the app. The intuition is to first compute a set of class variables $vars$ that are *not* initialized through a *default initializer*. The *default initializer*s are the methods that get automatically invoked whenever a class or activity gets created, *e.g.*, the life cycle methods of an activity, class constructors, *etc.* These variables $vars$ are our target candidates, since they are defined and accessed only through callbacks. Next, for every such variable $var \in vars$, COLUMBUS searches for callback pairs (cb_1, cb_2) where one of them *read*s (`R`) var, and the other *write*s (`W`) var. The output of this phase will be a set of variables with their dependent callback pairs. For the example in Figure 2.1, the output will be {`mSelectedFile`, (`'R'`, `onClick`), (`'W'`, `onListItemClick`)}.

These dependency pairs are used as feedback during the exploration phase detailed in Section 2.3.4. In order to accelerate crash discovery, COLUMBUS implements a weighted-score based exploration strategy, which initially prioritizes executing callbacks that write to variables over the callbacks that read from the same variables—inducing the dependency violating callback invocation orders. However, during the exploration, COLUMBUS dynamically adjusts the scores, *e.g.*, penalizes the callbacks that frequently result in a

crash, or prioritizes the callbacks that are executed less frequently, in order to explore newer or less explored program paths as well.

2.3.4 Callback-guided exploration

To explore an app under test, we first statically obtain its callbacks (Section 2.3.1), their dependencies (Section 2.3.3), and the primitive argument values (Section 2.3.2). Then, COLUMBUS spawns the app, dynamically instruments it to inspect the app heap, and starts exploring its functionalities. COLUMBUS invokes a callback whenever an instance of the activity, or the class defining the callback appears in the app's heap. If the callback expects reference type arguments, COLUMBUS then generates such argument objects using the strategy detailed in Section 2.3.2. Algorithm 2 gives an overview of our app exploration strategy. COLUMBUS's exploration strategy is composed of the following components:

Activity monitor. As the app is being explored, two kinds of entities get created, or destroyed in the heap: **(i)** activities and related UI element objects, and **(ii)** regular class objects, as the side-effect of calling a callback that instantiates the class. The activity monitor records such events by monitoring the invocation of the lifecycle callbacks of the activities, and the class constructors. For example, invocation of `onCreate()` signals an activity creation, and `onDestroy()` is invoked when an activity is destroyed. The activity monitor maintains an activity stack $\mathcal{S}$ by pushing an activity to $\mathcal{S}$ when a new activity is created, and popping an activity off $\mathcal{S}$ when it is destroyed. Therefore, the most recently created activity, which we call as the *live* activity, always remains at the top of $\mathcal{S}$.

The app is explored in a depth-first manner, and runs in continuous *cycles*. For a live activity *act*, the activity monitor retrieves all the class objects $newClasses$ created in the app heap (Line 18), passes it on to the *selector* for choosing the next callback *cb*, which is

Algorithm 2: Callback driven exploration

```
1  Function CallbackExploration
       Input  : Application callbacks AC, their dependencies Dep, class hierarchical information
                CH_f and CH_a, duration t
       Output : Crash dumps crashes
2      crashes ← ∅, explored ← {}, testingCycle ← 0
3      CbW ← ∅        // callback weights
4      ClW ← ∅        // class weights
5      foreach callback cb ∈ AC do
6          cl ← GetclassDefiningMethod(cb)
7          CbW ← CbW ∪ (cl, cb, 0.0)
8          ClW ← ClW ∪ (cl, 0.0)
9      end
10     while until t is reached do
11         spawnApp ()
12         testingCycle ← testingCycle + 1
13         foreach callback cb ∈ AC do
14             explored[cb] ← false
15         end
16         while until no new activity left to explore do
17             act ← getLiveActivity()
18             newClasses ← getNewClasses(act, explored)
19             if newClasses = ∅ then
20                 RemoveActivity(act)
21                 go to Line 16
22             end
23             cl ← getNextClass(newClasses ∪ act, ClW, Dep)
24             cb ← getNextCallback(cl, explored, CbW, Dep)
25             if cb = ∅ then
26                 explored ← explored − (cb, false) ∪ (cb, true)
27                 go to Line 16
28             end
29             allargs ← generateArguments(cl)
30             foreach args ∈ allargs do
31                 inst ← getInstance(cl)
32                 newCrash ← ExecuteCallback(inst, cb, args)
33                 if newCrash ≠ ∅ then
34                     crashes ← crashes ∪ newCrash
35                     UpdateAndPenalizeWeights(ClW, CbW, cl, cb)
36                     restartApp () and go to Line 10
37                 end
38                 else
39                     UpdateWeights(ClW, CbW, cl, cb)
40                 end
41             end
42         end
43     end
44     return crashes
```

then executed by the *executor*. The function `getNewClasses()` returns only those classes for which at least one callback is still unexplored. If a callback creates a new live activity act', the activity monitor puts act on hold, and switches to act'. When all the callbacks of an activity or its associated classes have been executed, the activity monitor destroys the activity, removes it from $\mathcal{S}$ (Lines $19 - -22$), and starts exploring the next live activity. One testing cycle ends, and the next one begins when $\mathcal{S}$ becomes empty.

Selector. The selector module receives the candidate classes $newClasses$ to be explored from the activity monitor, and chooses a callback cb to be executed next (Line 24). While choosing cb, it considers the class weights ClW, callback weights CbW, inter-callback dependencies Dep, and the visited status $explored$ of the callbacks. The $explored$ map is cleared when a testing cycle begins. All the weights are initially set to zero, and are dynamically adjusted during the exploration based on how frequently the classes and the callbacks have been explored. Similarly, when a callback is explored, the $explored$ map is updated (Line 26).

To choose a callback, the selector employs multiple strategies in the following order: **(i)** In the beginning, when none of the callback is explored, the selector uses Dep to choose the callback cb with the read (`R`) dependency, and its defining class cl. **(ii)** The selector consults the $explored$ map to prioritize unexplored callbacks over the explored ones. **(iii)** A class or callback with lower weight (ClW or CbW) has been explored the least; therefore it is prioritized next for execution. The tie among multiple unexplored classes, or callbacks with the same weight is broken randomly.

Executor. The executor executes the callback selected by the selector. The executor searches the app heap for an instance of a class, or an activity that overrides the callback (Line 31). If an instance is found, the executor generates the arguments for the callback respecting their types (Section 2.3.2). However, an argument can have multiple possible values executing different paths (primitive), or depending on the availability of objects in

the heap (reference). The executor, therefore, schedules the callback for execution for each combination of such inferred values. After each execution, the class weight for a class *cl* and the callback weight for a callback *cb* are updated as shown in Figure 2.5.

$$
\begin{aligned}
CbW_{cb} \;\; :- \;\; & CbW_{cb} + \tfrac{ex_t}{sch} \\
& sch \leftarrow \text{number of scheduled executions of } cb \\
& ex_t \leftarrow \text{number of executions of } cb \text{ at time } t \\
\\
ClW_{cl} \;\; :- \;\; & avg(CW_{cb}) \; \forall cb \in cl
\end{aligned}
$$

Figure 2.5: New class and callback weights after each execution

Intuitively, the executor updates the weights to reflect what percentage of callbacks are executed with respect to the total number of possible invocations—since a crash, or a creation of new activity may interrupt the processing of the rest of the scheduled executions. The class weights are accordingly adjusted such that the least explored class, and its callbacks are prioritized to be executed the next time the activity comes live.

Crash detector. After the execution of a callback, the crash detector monitors whether it results in a crash of the app. We do not want to rediscover the same crash repeatedly. Therefore, if a crash happens, the `UpdateAndPenalizeWeights()` (Line 35) function updates the class weights to deprioritize the callback *cb*, and its defining class *cl*—the callback weight CbW_{cb} is increased by δ (an empirically determined constant), and accordingly the class weight ClW_{cl} is adjusted. The idea is to gradually increase the callback weight in order to account for the case when *only* a specific set of argument values results in a crash, and all other values should still be able to explore the callback. Therefore, instead of not choosing the callback at all, the selector deprioritizes the callback for some time.

2.4 Evaluation

In our evaluation, we aim at answering the following research questions: **RQ1.** How does COLUMBUS compare with the state-of-the-art testing tools in terms of both code coverage and discovered crashes? **RQ2.** How effective is COLUMBUS in finding crashes in popular, real-world apps? **RQ3.** What is the benefit of leveraging dependency feedback?

2.4.1 Experimental setup

Dataset. To answer **RQ1** and **RQ3**, we used AndroTest [40], a collection of 68 apps. This dataset has become the de facto standard benchmark for Android app testing, and it has been used in the evaluation of a large number of tools [40, 17, 44, 41, 45, 46, 47, 48, 49, 16, 50, 51]. However, we had to remove 8 apps that were not fully compatible with Android 9 (which is the environment we used for COLUMBUS). For example, the `ListView` in the `netcounter` app does not appear in Android 9. Therefore, we used the remaining 60 apps for all our experiments.

For **RQ2**, we created a dataset of popular, real-world apps. We will refer to this dataset as the *real-world* dataset. To build this dataset, we first compiled a list of Google Play Store [52] apps with a minimum of 500,000 installs and a user rating of at least 4.5 stars. Then, we collected first 140 apps compatible with FRIDA instrumentation. As we show in Table 2.2, these apps are quite diverse and belong to 14 broad categories.

Environment. Our experiments were conducted on a system with an Intel(R) Core(TM) i9-10885H @ 2.40GHz processor (16 cores), 128GB of memory, and 1TB of solid-state drive (relevant for the snapshot save and restore mechanism used by TIMEMACHINE), running a 64-bit Ubuntu 20.04 operating system. For testing, we used 8 Google Pixel 3a phones running Android 9 (Pie, API level 28), with the Internet and Bluetooth connectivity enabled. We did not create any accounts for those apps that allow user logins. We ran each tool for 3

hours on each app, repeated each experiment 5 times, and averaged out the results to minimize the effect of any inherent randomness. Before testing each app, we first brought the phones to a *clean-slate* state by wiping its `sdcard` contents, and then pushed the `sdcard` files used by STOAT in their experiment to the phones. All the tools except TIMEMACHINE, which requires a virtual machine (VM) to operate, were tested on real hardware (phone).

Pre-exploration. Before the dynamic exploration could begin, COLUMBUS prepares an app by running the first three static pre-processing phases. We provide relevant results for the 60 apps of the AndroTest dataset: The *callback discovery* module identified a total of 30,682 and 4,991 callbacks in the Android framework and the apps, respectively. Out of 4,991 app callbacks discovered, 1,566 callbacks had at least one primitive argument, thus necessitating the invocation of the *argument generation* module. With a timeout of 5 minutes, the argument generation succeeded for 1,332 callbacks, while it timed out for the remaining 234 callbacks. Additionally, 4,147 callbacks have at least one reference type argument, and in total 4,857 reference type arguments. Out of them, 4,650 objects were always found on the heap, and the remaining 207 objects needed to be created. Finally, the *callback dependency* module discovered a total of 2,456 dependency relations between 975 variables across all the apps.

Coverage and crash collection. We used EMMA [53] to collect statement coverage. The coverage data was collected every minute for all tested tools. EMMA injects its own instrumentation code into the apps. Unfortunately, its coverage reports do include coverage data from its own packages, which can either inflate, or deflate the overall coverage. Therefore, we excluded EMMA-specific classes from the coverage calculation.

We detect crashes by parsing **(i)** LOGCAT [54] logs fetched by the log watcher, a long-running process that streams logs from the devices (phones) in real-time, and **(ii)** logs of the crashes captured by the FRIDA server. We used the widely adopted practice of computing the stack hash to determine the *uniqueness* of crashes. Crashes that do not contain

the app's package name were filtered out. For FRIDA reports, we occasionally observed that certain crashes that originate from the dynamic instrumentation contain an app's package name. Therefore, we manually inspected and removed those irrelevant crashes after the initial package-name-based filtering. Then, we normalized the stack traces for the remaining crashes by removing irrelevant and ephemeral information, *e.g.*, timestamp, process id (PID), *etc.* Finally, we compute hashes over these sanitized stack traces.

Implementation. We implemented the first three phases of our analysis, *viz.*, callback identification, callback dependency discovery, and primitive argument generation using the ANGR [55] binary analysis framework. All these phases are performed offline, before the testing begins on the device. For exploration, the final phase, we leveraged the FRIDA [56] dynamic instrumentation toolkit.

2.4.2 Experimental results

Performance on benchmark apps.

To investigate how our technique performs with respect to prior work, we use the AndroTest benchmark apps. Specifically, we compared the achieved code coverage and the number of crashes found by COLUMBUS with the state-of-the-art model-based techniques STOAT [17] and APE [20], checkpoint-based technique TIMEMACHINE [41], and callback-driven technique EHBDROID [21]. Unfortunately, we could not make the publicly available version of EHBDROID work on our test apps due to the incompatibility of their instrumentation module with our test subjects. Instead, we implemented their testing strategies by modifying COLUMBUS in three ways: **(i)** we consider only those 58 callbacks supported by EHBDROID, **(ii)** we disabled dependency and crash guidance, and **(iii)** we restricted primitive argument values to those used by EHBDROID instead of the values computed by our argument generation module.

Apps	Line coverage						Crashes					
	ST	EH	AP	TM	CB	CB_{wd}	ST	EH	AP	TM	CB	CB_{wd}
mileage	38	23	58	40	60	57	2	0	15	9	4	4
bomber	61	56	66	97	88	87	0	0	0	0	0	0
mirrored	31	16	38	46	47	47	0	0	0	1	1	1
batterydog	59	5	72	73	72	72	0	0	0	1	0	0
triangle	90	91	90	91	91	91	0	0	0	0	1	1
translate	46	29	48	48	49	49	1	1	1	0	1	1
anymemo	26	18	50	42	52	46	2	1	6	6	7	7
zooborns	18	17	19	25	26	26	3	0	3	3	1	1
qsettings	40	23	50	40	47	46	1	1	1	0	1	0
wchart	57	24	32	51	85	83	2	1	0	0	3	3
addi	17	16	21	19	18	18	1	0	8	1	3	3
LNM	49	3	34	48	50	50	4	0	4	7	2	1
gestures	32	32	32	50	78	78	0	0	0	0	0	0
MNV	35	13	64	42	68	68	2	1	4	4	1	1
wikipedia	24	21	25	31	19	19	0	0	0	0	0	0
dialer	66	53	65	40	73	73	1	1	1	3	2	2
photost	24	9	26	28	12	12	2	1	1	3	3	3
battery	92	55	55	93	88	88	0	0	0	3	0	0
aCal	18	8	28	29	22	19	3	0	5	3	3	1
tomdroid	55	24	57	53	61	59	0	0	4	0	2	2
RMP	82	87	83	65	92	92	1	0	0	1	2	2
SpriteText	62	63	62	63	61	59	0	0	0	0	0	0
LPG	63	37	89	82	0	0	0	0	0	0	0	0
ringdroid	0	40	42	23	47	47	1	2	4	2	2	2
sftp	11	5	15	12	18	18	0	0	0	0	3	1
PWMG	3	6	7	16	6	6	0	1	0	0	2	2
fbubble	49	49	56	82	74	72	0	0	0	0	3	3
myexp	55	1	33	46	65	63	0	0	0	1	7	7
sanity	13	8	26	27	36	35	1	0	2	1	2	1
SMT	87	2	87	63	87	85	0	0	0	0	0	0
alogcat	65	33	73	79	60	53	0	0	0	0	2	2
worldclock	97	90	98	94	95	95	1	1	0	1	2	2
mlife	87	35	86	84	92	92	0	0	0	0	2	2
lbuilder	22	28	28	26	37	35	0	1	0	0	4	4
CDT	63	31	65	85	87	87	0	0	0	0	0	0
bites	26	15	42	36	54	54	2	0	5	8	3	3
multisms	40	26	74	57	78	78	0	1	0	1	1	1
yahtzee	69	3	46	6	51	46	1	0	3	1	3	3
nectroid	40	27	44	38	46	46	0	0	0	2	2	2
anycut	70	12	71	71	66	66	0	2	0	0	3	3
PMM	66	27	62	56	65	62	4	0	11	3	4	4
manpages	40	20	54	77	78	74	0	0	0	1	3	3
zoffcc	18	15	16	20	16	16	3	0	4	1	4	4
amazed	62	64	76	52	84	84	0	0	1	1	1	1
alarmclock	72	15	76	68	71	71	6	0	4	4	5	5
hndroid	13	5	11	8	15	15	0	1	0	2	2	2
sboard	100	58	100	100	100	100	0	0	0	0	0	0
hotdeath	16	63	73	75	80	76	1	3	2	0	5	5
dalvik-exp	23	6	72	70	64	64	1	0	5	3	4	4
jamendo	10	13	28	9	30	30	5	3	0	0	5	5
importcont	57	2	53	42	78	74	0	0	0	0	1	1
blokish	36	35	49	52	45	45	0	0	2	0	2	2
Book-cat	4	4	33	35	38	38	0	1	2	4	4	0
Templaro	55	76	87	60	86	83	0	1	0	2	3	3
DAC	53	48	76	88	94	91	0	0	0	0	0	0
Agrep	37	8	58	63	61	58	0	0	7	2	7	7
Syncmypix	15	18	21	25	26	26	1	1	0	1	3	3
tippytipper	72	9	86	84	89	89	0	0	0	0	2	2
WHAMS	80	0	77	69	79	79	0	0	0	1	1	1
A2dp	29	14	40	45	47	42	6	0	6	0	3	3
Avg/Sum	46	27	53	52	58	57	58	25	111	87	137	126

Table 2.1: Coverage and the number of crashes reported by all the tools in the AndroTest dataset. ST: STOAT, EH: EHBDROID, AP: APE, TM: TIMEMACHINE, CB: COLUMBUS, CB_{wd}: COLUMBUS without dependency feedback

In Table 2.1, we present the statement coverage achieved as well as the crashes triggered by all tools on the benchmark apps.

Coverage. We find that COLUMBUS achieves higher code coverage than STOAT, EHBDROID, APE and TIMEMACHINE for 45, 55, 41, and 41 apps, respectively. Moreover,

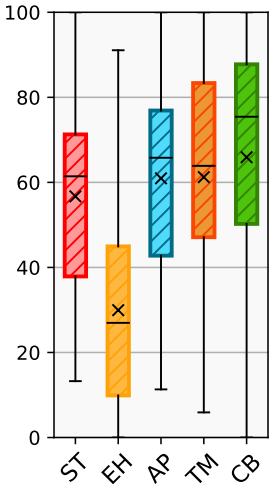

Figure 2.6: <1K (30)

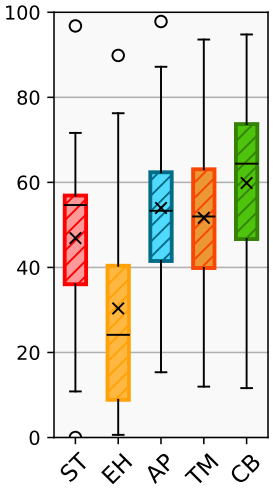

Figure 2.7: [1K,3K) (17)

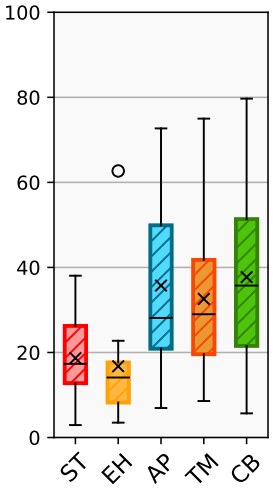

Figure 2.8: ≥3K (13)

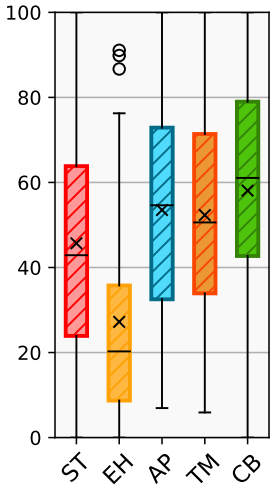

Figure 2.9: all (60)

Figure 2.10: Coverage (Y-axis) achieved on AndroTest, grouped by app size (Lines of Code). Number of apps in a size group is indicated in parentheses. 'x' denotes the mean of a boxplot

COLUMBUS achieves the best coverage in 36 apps, followed by TIMEMACHINE (16 apps), APE (10 apps), STOAT (5 apps), and EHBDROID (2 apps). To gain an overall view of the tools' performances, we report the average code coverage, achieved by each tool across all apps, in the last row of Table 2.1. As can be seen, COLUMBUS attains the highest (58%) coverage on average, followed by APE (53%), TIMEMACHINE (52%), STOAT (46%), and EHBDROID (27%). Figure 2.12 shows the progression of coverage over time for all the tools averaged across all the benchmark apps. Starting from the 5th minute, the coverage achieved by COLUMBUS exceeds other tools. Until approximately the 20th minute, the coverage increases at a fairly fast rate, after that, it starts to slow down. Further, the boxplot in Figure 2.10 shows the *spread* of the coverage achieved by all the tools grouped by the size of the apps. We use group sizes identical to the ones used in previous work [41]. As the figure shows, COLUMBUS exhibits significant improvement over other tools in terms of coverage for all size groups.

The improvement in coverage for COLUMBUS can be attributed to its systematic

exploration of the callbacks. While UI-based techniques struggle to generate complex events and appropriate user input, COLUMBUS sidesteps this problem by directly calling the callbacks and supplying argument values (computed by the argument generation module) that are likely to explore additional code paths. In addition, the crash-guidance feedback helps COLUMBUS to make the best use of the time-budget by preventing the exploration from getting stuck at individual crashes for a long time.

Figure 2.11 shows a code snippet from the `RandomMusicPlayer` app from AndroTest. This example shows an interesting case where COLUMBUS naturally enjoys clear benefits over previous, more "heavyweight" techniques that use symbolic execution [47], and other UI-testing tools. To explore all the branches (`if` conditions), a UI-based tool would need to `click` on all corresponding buttons, which is challenging. ACTEVE [47] solves this problem by concolically executing the app together with an instrumented version of the Android framework. Since, in our case, COLUMBUS introspects the app heap to retrieve live objects, we observed the coverage of this app quickly going up, because COLUMBUS invokes the `onClick` callback with all the button `Views` already present in the heap.

To better understand the challenges COLUMBUS faces during exploration, we manually examined 10 of those apps where COLUMBUS did not achieve the best coverage. We summarize our findings next: **(i)** For callbacks where the symbolic execution timed out, the *argument generation* module could not return any useful value. As a result, COLUMBUS fell back to its default strategy of trying out random argument values, which negatively affected the coverage. **(ii)** There exist callbacks that are *stateful*. That is, the application logic is conditioned on `class` variables. Note that COLUMBUS is not state-aware, therefore this challenge is orthogonal to what COLUMBUS aims to solve. **(iii)** For unconstrained callback arguments, we use random values from a predefined list, which might be ineffective. For instance, the `yahtzee` app lists the game moves in a drop-down list. A move can be chosen by the `arg2` argument (unconstrained) of

```
1  public void onClick(View target) {
2    // Send intent according to the button clicked
3    if (target == mPlayButton) {
4       startService(new Intent(MusicService.ACTION_PLAY));
5    } else if (target == mPauseButton) {
6       startService(new Intent(MusicService.ACTION_PAUSE));
7    } else if (target == mSkipButton) {
8       startService(new Intent(MusicService.ACTION_SKIP));
9    } else if (target == mRewindButton) {
10     startService(new Intent(MusicService.ACTION_REWIND));
11   } else if (target == mStopButton) {
12      startService(new Intent(MusicService.ACTION_STOP));
13   } else if (target == mEjectButton) {
14      showUrlDialog();
15   }
16 }
```

Figure 2.11: Code snippet (redacted) from RandomMusicPlayer

the `onItemSelected(_, _, arg2, _)` callback, which then looks up the appropriate UI object using that argument. Many such values of `arg2` that we supply could be invalid, while UI-based techniques can "blindly" `click` on the list item without being aware of the valid values of that argument.

Crashes. COLUMBUS found a total of 153 crashes. After excluding the potential false positives, the total number of crashes become 137 (Table 2.1). As presented in Table 2.3, COLUMBUS found crashes of 16 different types in 49 out of 60 apps in the AndroTest dataset. Compared to STOAT, EHBDROID, APE, and TIMEMACHINE, COLUMBUS discovered 4.42, 5.48, 1.23, and 1.57 times more crashes, respectively. To acquire a better understanding of how the tools perform on individual apps, we calculated the number of apps for which each tool discovers the most number of crashes. While STOAT, EHBDROID, APE, and TIMEMACHINE finds the most crashes in 14, 10, 25 , and 21 apps, respectively, COLUMBUS performs the best for the highest (45) number of apps.

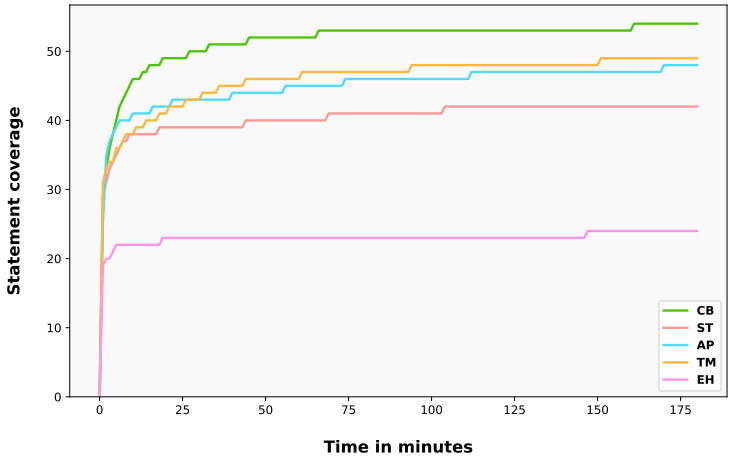

Figure 2.12: Progression of coverage over time by all the tools on the AndroTest dataset. Tool codes are similar to Table 2.1

False positive analysis. Our strategy of invoking callbacks directly, sometimes with artificially-prepared arguments, can potentially lead to false positives (FP), *i.e.*, generate spurious crashes that cannot be triggered when the app is normally exercised from the UI. Since STOAT, APE, and TIMEMACHINE are UI-driven testing tools, they always generate legitimate crashes. For COLUMBUS, we identify two potential reasons for FPs and quantify their prevalence.

(i) Disabled UI elements. Since COLUMBUS does not access the UI state of the app, it may (incorrectly) invoke a callback cb_d associated with a widget W, which is disabled at the time of invocation. If such a callback cb_d exists in an app, then there exists another callback cb_e that calls `W.setEnabled()` to enable the widget. We found that only 71 (cb_e) out of 4,991 callbacks in our benchmark apps contain such calls. Now, `setEnabled` calls from inside the lifecycle callbacks are not problematic. Because, the latter is called by the Android framework, which enables the respective UI elements as part of the initialization of the app. Among those 71, only 4 callbacks are non-lifecycle ones, which is negligible with respect to the total number of callbacks.

(ii) Uninitialized nested object argument. If a callback expects an object argument of `class A` that we do not find in the heap, we create an instance a by

invoking the class constructor C. However, instances created in this way may be partially uninitialized. Suppose, A contains a field $A.b$ of `class B`, which C leaves uninitialized. If the callback attempts to access $A.b$, then it will result in a `NullPointerException`. This is a spurious crash, because when the app is exercised from the UI, the framework would invoke the callback with a correctly constructed object. In case of the benchmark apps, we needed to create object arguments for only 207 (4.15%) out of 4,991 callbacks. Unfortunately, there is no straightforward way to estimate further how many of these callbacks require nested object arguments. Even then, since we already invoke object creation for a reasonably small number of callbacks, that makes the probability of such FPs minimal.

To investigate into our potential sources of FPs, we first collected all 55 crashes that are found only by COLUMBUS, but not by any of those tools. Then, we manually verified those reports to determine potential FPs. We call a report *legitimate*, if we can reproduce a crash with the same stack trace by exercising the app from the UI. To do that, we collected a sequence of callback invoked immediately before the crash from our tool's output log, and also reviewed the relevant part of the source code to seek further guidance. If we failed to reproduce the crash within a reasonable number of tries, we flagged the report as FP. Note that, this estimate is conservative and best-effort, because it includes true crash reports that we could not reproduce because of Android apps' inherent statefulness. At the end, we failed to reproduce 16 crashes out of total 153 crashes, which, even in the worst case, translates to a mere 10.46% FP rate. We argue that this amount of FPs is acceptable in practice, given the benefits (extra crashes, coverage) that our approach brings.

RQ1: Compared to the state-of-the-art tools, COLUMBUS attains the highest coverage on average (58%), and discovers the most number of crashes (137) on the AndroTest dataset.

Category	Count
Education	27
Games	26
Personalization	18
Tools	17
Multimedia	11
Photography	4
Lifestyle	7
Health & Fitness	4
Food & Drink	4
Entertainment	6
Travel & Local	6
Business	2
Productivity	4
Others	4
Total	140

Table 2.2: Real-world app categories

ID	Exception type	A	R
1	NullPointerException	52	22
2	IllegalStateException	16	26
3	ArrayIndexOutOfBoundsException	7	4
4	IndexOutOfBoundsException	10	2
6	CursorIndexOutOfBoundsException	10	-
7	UnsatisfiedLinkError	6	-
8	RuntimeException	1	2
9	IllegalArgumentException	15	4
10	ClassCastException	1	2
12	StaleDataException	3	-
13	ActivityNotFoundException	8	6
14	SQLiteDoneException	1	-
15	NumberFormatException	1	-
16	App Exceptions	6	2
	Total	137	70

Table 2.3: Crashes found by COLUMBUS. A: AndroTest, R: Real-world dataset

Performance on real-world apps.

To understand the practicality of our approach, we tested COLUMBUS on the real-world dataset. In line with the previous approaches [44, 17, 41], we only considered the number of crashes discovered by our tool for this evaluation.

Crashes. As shown in Table 2.3, we discovered a total of 70 crashes of 9 different types in 54 out of 140 apps, where `IllegalStateException` (37.14%) and `NullPointerException` (31.43%) are the most prevalent ones.

RQ2: COLUMBUS is able to find 70 crashes in 54 out of 140 real-world Play Store apps, belonging to 14 categories.

Effectiveness of dependency feedback.

To show the effectiveness of the dependency feedback, we performed an ablation study by comparing COLUMBUS with COLUMBUS $_{\text{wd}}$, a modified version of our tool that runs without the dependency feedback. Table 2.1 presents the results of this experiment on the AndroTest dataset.

While the coverage attained by both COLUMBUS and COLUMBUS $_{\text{wd}}$ are comparable,

the latter finds -3 fewer crashes than the former in 5 apps. By manually inspecting those apps—`Book-cat`, `qsettings`, `sanity`, `sftp`, and `aCal`, we can confirm that the additional crashes are correlated with the number of dependency relations discovered. In other words, due to higher than average (41 dependencies/app) number of dependencies being present in those apps, the dependency feedback could indeed help COLUMBUS in triggering more crashes. In addition, COLUMBUS achieved better coverage than any other tool for the first four apps.

RQ3: The dependency feedback used by COLUMBUS is useful for triggering crashes in apps, particularly for those ones with large amount of inter-callback dependencies.

2.5 Limitations

Inferring correct value of the object fields. Currently, our *argument generation* module can only infer the correct values of the primitive arguments. However, it can be extended to support object arguments as well. Consider the callback: `onKeyDown (int keyCode, KeyEvent event)`, which gets called when a key down event occurs. Now, `event.getUnicodeChar()` API returns the Unicode character c generated by that key event. If a callback has paths conditioned on c, we can infer its correct values by symbolizing the return value of the API. The inferred values can be used during testing to either dynamically set the correct value of the appropriate field of the `event` argument, or 'hook' the `getUnicodeChar()` API to alter its return value—exercising more paths in effect.

Creating values for login. There are Android apps which requires a userid and password to login first before one can explore its functionality. COLUMBUS in its current shape can not detect such a login prompt, and enter the username and password automatically

to explore such an app. However, this is a limitation that we share with the existing state-of-the art tools, and an interesting direction for future work.

2.6 Related work

Random. Random testing based techniques such as MONKEY [14] delivers random events. DYNODROID [45], in addition, considers system-level events, and monitors which events have registered listeners in the app to prioritize certain events depending on the context. PUMA [57] presents an automation framework that has support for custom dynamic exploration strategies. However, random testing strategies, though popular, often get stuck in a "local optima," making no further progress.

Model-based. Model-based testing approaches guide the exploration of the app by deriving a model of the app's UI. Though some techniques require this model to be provided manually [58, 59, 60], others reconstruct the UI model using dynamic app exploration [61, 62, 17, 63, 64, 65]. Other techniques also perform model abstraction via identifying the structural similarities between different layouts [66], model refinement by merging several UI interaction [20], and state recovery using snapshotting [41]. Model based testing techniques oftentimes suffer from state explosion if there are too many states in the app. Therefore, they need to strike a balance between model completeness and scalability.

Symbolic execution-based. Anand *et. al.* [47] concolically executes both the Android framework and the entire app, which is precise, but not scalable. In contrast, COLUMBUS does symbolic execution only within a callback to strike a balance between precision, and scalability. Another approach [67] starts the symbolic exploration in reversed order from the target blocks, and obtains the sequences of events to reach these targets. Additionally, several other techniques were introduced for the symbolic execution of the apps that

include libraries as well [68, 69].

Hybrid. Similar to COLUMBUS, several approaches also employ hybrid techniques, *i.e.*, combination of static and dynamic strategies, for app exploration. In particular, [70, 71, 72, 73, 74] reconstruct the app model statically, followed by dynamic exploration. Other techniques use static analysis to discover dependencies between different application components, and use it during the dynamic exploration [70, 75, 67, 76, 74, 77]. Another guided exploration technique CAR [78] uses a static constraint analysis to keep the symbolic execution scalable and obviate the need for whole program symbolic execution. In contrast, COLUMBUS aims to maximize coverage similar to other app testing tools limiting the scope of the symbolic execution only within the callback and sets up the environment in an under-constrained manner. Moreover, during the dynamic exploration, COLUMBUS uses a type-guided object matching to supply an existing, well-formed object to the callback. Whereas, CAR resorts to a refinement-based construction of heap objects, guided by a crash-oracle. A crash resulting from a malformed object acts as a 'hint' to fix the shape of the object. EHBDROID [21] instruments the app statically to include callback invocations within the app code in order to invoke them directly. However, their technique is not generic, and suffers from limitations as discussed before.

2.7 Conclusion

In this chapter, we introduce COLUMBUS, a callback-driven testing technique that addresses the limitations of prior approaches. COLUMBUS automatically identifies callbacks by analyzing both the Android framework and the app under test. It utilizes a combination of under-constrained symbolic execution and type-guided dynamic heap introspection to generate valid and effective inputs for callbacks, reducing the reliance on human involvement. Furthermore, COLUMBUS incorporates two novel feedback mechanisms, data

dependency, and crash-guidance, to enhance testing effectiveness. These mechanisms increase the likelihood of triggering crashes and maximize coverage during testing. The evaluation of COLUMBUS demonstrates its superiority over state-of-the-art model-driven, checkpoint-based, and callback-driven testing tools in terms of crashes detected and coverage achieved. COLUMBUS offers a promising solution for automated testing of Android apps, providing improved efficiency and effectiveness compared to existing techniques.

Chapter 3

Sailfish: Vetting Smart Contract State-Inconsistency Bugs in Seconds

In this chapter, I present Sailfish, a highly scalable tool that is aimed at automatically identifying state-inconsistency bugs in smart contracts. To tackle the scalability issue associated with statically analyzing a contract, Sailfish adopts a hybrid approach that combines a light-weight Explore phase, followed by a Refine phase guided by our novel *value-summary analysis*, which constrains the scope of storage variables. Our Explore phase dramatically reduces the number of relevant instructions to reason about, while the value-summary analysis in the Refine phase further improves performance while maintaining the precision of symbolic evaluation. Given a smart contract, Sailfish first introduces an Explore phase that converts the contract into a *storage dependency graph* (SDG) G. This graph summarizes the side effects of the execution of a contract on storage variables in terms of read-write dependencies. State-inconsistency vulnerabilities are modeled as graph queries over the SDG structure. A vulnerability query returns either an empty result—meaning that the contract is not vulnerable, or a potentially vulnerable subgraph g inside G that matches the query. In the second case, there are

two possibilities: either the contract is indeed vulnerable, or g is a false alarm due to the over-approximation of the static analysis.

To prune potential false alarms, SAILFISH leverages a REFINE phase based on symbolic evaluation. However, a conservative symbolic executor would initialize the storage variables as *unconstrained*, which would, in turn, hurt the tool's ability to prune many infeasible paths. To address this issue, SAILFISH incorporates a light-weight *value-summary analysis* (VSA) that summarizes the value constraints of the storage variables, which are used as the pre-conditions of the symbolic evaluation. Unlike prior summary-based approaches [79, 80, 81] that compute summaries path-by-path, which results in full summaries (that encode all bounded paths through a procedure), leading to scalability problems due to the exponential growth with procedure size, our VSA summarizes *all paths* through a finite (loop-free) procedure, and it produces compact (polynomially-sized) summaries. As our evaluation shows, VSA not only enables SAILFISH to refute more false positives, but also scales much better to large contracts compared to a classic summary-based symbolic evaluation strategy.

We evaluated SAILFISH on the entire data set from ETHERSCAN [11] (89,853 contracts), and showed that our tool is efficient and effective in detecting state-inconsistency bugs. SAILFISH significantly outperforms all five state-of-the-art smart contract analyzers we evaluated against, in the number of reported false positives and false negatives. For example, on average SAILFISH took only 30.79 seconds to analyze a smart contract, which is 31 times faster than MYTHRIL [27], and 6 orders of magnitude faster than SECURIFY [25].

3.1 Background

This section introduces the notion of the state of a smart contract, and provides a brief overview of the vulnerabilities leading to an inconsistent state during a contract's

execution.

Smart contract. Ethereum smart contracts are written in high-level languages like SOLIDITY, VYPER, *etc.*, and are compiled down to the EVM (Ethereum Virtual Machine) bytecode. Public/external methods of a contract, which act as independent entry points of interaction, can be invoked in two ways: either by a *transaction*, or from another contract. We refer to the invocation of a public/external method from outside the contract as an *event*. Note that events exclude method calls originated from inside the contract, *i.e.*, a method f calling another method g. A *schedule* $\mathcal{H}$ is a valid sequence of events that can be executed by the EVM. The events of a schedule can originate from one or more transactions. Persistent data of a contract is stored in the storage variables which are, in turn, recorded in the blockchain. The *contract state* $\Delta = (\mathcal{V}, \mathcal{B})$ is a tuple, where $\mathcal{V} = \{V_1, V_2, V_3, ..., V_n\}$ is the set of all the storage variables of a contract, and $\mathcal{B}$ is its balance.

State inconsistency (SI). When the events of a schedule $\mathcal{H}$ execute on an initial state Δ of a contract, it reaches the final state Δ'. However, due to the presence of several sources of non-determinism [82] during the execution of a smart contract on the Ethereum network, Δ' is not always predictable. For example, two transactions are not guaranteed to be processed in the order in which they got scheduled. Also, an external call e originated from a method f of a contract $\mathcal{C}$ can transfer control to a malicious actor, who can now subvert the original control and data-flow by re-entering $\mathcal{C}$ through any public method $f' \in \mathcal{C}$ in the same transaction, even before the execution of f completes. Let $\mathcal{H}_1$ be a schedule that does not exhibit any of the above-mentioned non-deterministic behavior. However, due to either reordering of transactions, or reentrant calls, it might be possible to rearrange the events of $\mathcal{H}_1$ to form another schedule $\mathcal{H}_2$. If those two schedules individually operate on the same initial state Δ, but yield different final states, we consider the contract to have a state-inconsistency.

Reentrancy. If a contract $\mathcal{A}$ calls another contract $\mathcal{B}$, the Ethereum protocol allows $\mathcal{B}$ to call back to any public/external method m of $\mathcal{A}$ in the same transaction before even finishing the original invocation. An attack happens when $\mathcal{B}$ reenters $\mathcal{A}$ in an inconsistent state before $\mathcal{A}$ gets the chance to update its internal state in the original call. Launching an attack executes operations that consume gas. Though, SOLIDITY tries to prevent such attacks by limiting the gas stipend to 2,300 when the call is made through `send` and `transfer` APIs, the `call` opcode puts no such restriction—thereby making the attack possible.

In Figure 3.1, the `withdraw` method transfers Ethers to a user if their account balance permits, and then updates the account accordingly. From the external call at Line 4, a malicious user (attacker) can reenter the `withdraw` method of the `Bank` contract. It makes Line 3 read a stale value of the account balance, which was supposed to be updated at Line 5 in the original call. Repeated calls to the `Bank` contract can drain it out of Ethers, because the sanity check on the account balance at Line 3 never fails. One such infamous attack, dubbed "TheDAO" [5], siphoned out over USD $50 million worth of Ether from a crowd-sourced contract in 2016.

Though the example presented above depicts a typical reentrancy attack scenario, such attacks can occur in a more convoluted setting, *e.g.*, *cross-function*, *create-based*, and *delegate-based*, as studied in prior work [24]. A *cross-function* attack spans across multiple functions. For example, a function f_1 in the victim contract $\mathcal{A}$ issues an untrusted external call, which transfers the control over to the attacker $\mathcal{B}$. In turn, $\mathcal{B}$ reenters $\mathcal{A}$, but through a different function f_2. A *delegate-based* attack happens when the victim contract $\mathcal{A}$ delegates the control to another contract $\mathcal{C}$, where contract $\mathcal{C}$ issues an untrusted external call. In case of a *create-based* attack, the victim contract $\mathcal{A}$ creates a new child contract $\mathcal{C}$, which issues an untrusted external call inside its constructor.

Transaction Order Dependence (TOD). Every Ethereum transaction specifies the upper limit of the *gas* amount one is willing to spend on that transaction. Miners choose

```
1 contract Bank {
2  function withdraw(uint amount){
3    if(accounts[msg.sender] >= amount){
4         msg.sender.call.value(amount);
5         accounts[msg.sender]  = amount;
6    }
7  }
8 }
```

Figure 3.1:

```
1 contract Queue {
2   function reserve(uint256 slot){
3     if (slots[slot] == 0) {
4         slots[slot] = msg.sender;
5     }
6   }
7 }
```

Figure 3.2:

Figure 3.3: In Figure 3.1, the `accounts` mapping is updated after the external call at Line 4 . This allows the malicious caller to reenter the `withdraw()` function in an inconsistent state. Figure 3.2 presents a contract that implements a queuing system that reserves slots on a first-come-first-serve basis leading to a potential TOD attack.

the ones offering the most incentive for their mining work, thereby inevitably making the transactions offering lower *gas* starve for an indefinite amount of time. By the time a transaction T_1 (scheduled at time t_1) is picked up by a miner, the network and the contract states might change due to another transaction T_2 (scheduled at time t_2) getting executed beforehand, though $t_1 < t_2$. This is known as Transaction Order Dependence (TOD) [83], or *front-running* attack. Figure 3.2 features a queuing system where an user can reserve a slot (Line 3, 4) by submitting a transaction. An attacker can succeed in getting that slot by eavesdropping on the gas limit set by the victim transaction, and incentivizing the miner by submitting a transaction with a higher gas limit. Refer to Section 3.3 where we connect reentrancy and TOD bugs to our notion of state-inconsistency.

3.2 Motivation

This section introduces motivating examples of state-inconsistency (SI) vulnerabilities, the challenges associated with automatically detecting them, how state-of-the-art techniques fail to tackle those challenges, and our solution.

3.2.1 Identifying the root causes of SI vulnerabilities

By manually analyzing prior instances of reentrancy and TOD bugs—two popular SI vulnerabilities (Section 3.1), and the warnings emitted by the existing automated analysis tools [24, 27, 25, 28], we observe that an SI vulnerability occurs when the following preconditions are met: **(i)** two method executions, or transactions—both referred to as *threads* (*th*)—operate on the same storage state, and **(ii)** either of the two happens—**(a) Stale Read (SR)**: The attacker thread th_a diverts the flow of execution to read a stale value from `storage(`v`)` before the victim thread th_v gets the chance to legitimately update the same in its flow of execution. The reentrancy vulnerability presented in Figure 3.1 is the result of a stale read. **(b) Destructive Write (DW)**: The attacker thread th_a diverts the flow of execution to preemptively write to `storage(`v`)` before the victim thread th_v gets the chance to legitimately read the same in its flow of execution. The TOD vulnerability presented in Figure 3.2 is the result of a destructive write.

While the SR pattern is well-studied in the existing literature [25, 24, 28, 84], and detected by the respective tools with varying degree of accuracy, the reentrancy attack induced by the DW pattern has never been explored by the academic research community. Due to its conservative strategy of flagging any state access following an external call without considering if it creates an inconsistent state, MYTHRIL raises alarms for a super-set of DW patterns, leading to a high number of false positives. In this work, we not only identify the root causes of SI vulnerabilities, but also unify the detection of both the patterns with the notion of hazardous access (Section 3.2).

3.2.2 Running examples

Example 1. The contract in Figure 3.4 is vulnerable to reentrancy due to destructive write. It allows for the splitting of funds held in the payer's account between two payees —

```
1  // [Step 1]: Set split of 'a' (id = 0) to 100(%)
2  // [Step 4]: Set split of 'a' (id = 0) to 0(%)
3  function updateSplit(uint id, uint split) public{
4     require(split <= 100);
5     splits[id] = split;
6  }
7
8  function splitFunds(uint id) public {
9     address payable a = payee1[id];
10    address payable b = payee2[id];
11    uint depo = deposits[id];
12    deposits[id] = 0;
13
14    // [Step 2]: Transfer 100% fund to 'a'
15    // [Step 3]: Reenter updateSplit
16    a.call.value(depo * splits[id] / 100)("");
17
18    // [Step 5]: Transfer 100% fund to 'b'
19    b.transfer(depo * (100 - splits[id]) / 100);
20 }
```

Figure 3.4: The attacker reenters `updateSplit` from the external call at Line 16 and and sets `splits[id] = 0`. This enables the attacker to transfer all the funds again to `b`.

`a` and `b`. For a payer with id `id`, `updateSplit` records the fraction (%) of her fund to be sent to the first payer in `splits[id]` (Line 5) . In turn, `splitFunds` transfers `splits[id]` fraction of the payer's total fund to payee `a`, and the remaining to payee `b`. Assuming that the payer with `id` = 0 is the attacker, she executes the following sequence of calls in a transaction – **(1)** calls `updateSplit(0,100)` to set payee `a`'s split to 100% (Line 5); **(2)** calls `splitFunds(0)` to transfer her entire fund to payee `a` (Line 16); **(3)** from the fallback function, reenters `updateSplit(0,0)` to set payee `a`'s split to 0% (Line 5); **(4)** returns to `splitFunds` where her entire fund is *again* transferred (Line 19) to payee `b`. Consequently, the attacker is able to trick the contract into double-spending the amount of Ethers held in the payer's account.

Example 2. The contract in Figure 3.5 is non-vulnerable (safe). The `withdrawBalance` method allows the caller to withdraw funds from her account. The storage variable `userBalance` is updated (Line 10) after the external call (Line 9). In absence of the `mutex`, the contract could contain a reentrancy bug due to the delayed update. However, the

`mutex` is set to `true` when the function is entered the first time. If an attacker attempts to reenter `withdrawBalance` from her fallback function, the check at Line 4 will foil such an attempt. Also, the `transfer` method adjusts the account balances of a sender and a receiver, and is not reentrant due to the same reason (`mutex`).

3.2.3 State of the vulnerability analyses

In light of the examples above, we outline the key challenges encountered by the state-of-the-art techniques, *i.e.*, SECURIFY [25], VANDAL [84], MYTHRIL [27], OYENTE [28], and SEREUM [24] that find state-inconsistency (SI) vulnerabilities. Table 3.2 summarizes our observations.

Cross-function attack. The public methods in a smart contract act as independent entry points. Instead of reentering the same function, as in the case of a traditional reentrancy attack, in a cross-function attack, the attacker can reenter the contract through any public function. Detecting cross-function vulnerabilities poses a significantly harder challenge than single-function reentrancy, because every external call can jump back to any public method—leading to an explosion in the search space due to a large number of potential call targets.

Unfortunately, most of the state of the art techniques cannot detect cross-function attacks. For example, the *No Write After Call* (`NW`) strategy of SECURIFY identifies a storage variable write (`SSTORE`) following a `CALL` operation as a potential violation. MYTHRIL adopts a similar policy, except it also warns when a state variable is read after an external call. Both VANDAL and OYENTE check if a `CALL` instruction at a program point can be reached by a recursive call to the enclosing function. In all four tools, reentrancy is modeled after The DAO [5] attack, and therefore scoped within a single function. Since the attack demonstrated in Example 1 spans across both the `updateSplit` and `splitFunds`

methods, detecting such an attack is out of scope for these tools. Coincidentally, the last three tools raise alarms here for the wrong reason, due to the over-approximation in their detection strategies. SEREUM is a run-time bug detector that detects cross-function attacks. When a transaction returns from an external call, SEREUM write-locks all the storage variables that influenced control-flow decisions in any previous invocation of the contract during the external call. If a locked variable is re-written going forward, an attack is detected. SEREUM fails to detect the attack in Example 1 (Figure 3.4), because it would not set any lock due to the absence of any control-flow deciding state variable [1].

Our solution: To mitigate the state-explosion issue inherent in static techniques, SAILFISH performs a taint analysis from the arguments of a public method to the `CALL` instructions to consider only those external calls where the destination can be controlled by an attacker. Also, we keep our analysis tractable by analyzing public functions in *pairs*, instead of modeling an arbitrarily long call-chain required to synthesize exploits.

Hazardous access. Most tools apply a conservative policy, and report a read/write from/to a state variable following an external call as a possible reentrancy attack. Since this pattern alone is not sufficient to lead the contract to an inconsistent state, they generate a large number of false positives. Example 1 (Figure 3.4) without the `updateSplit` method is *not* vulnerable, since `splits[id]` cannot be modified any more. However, MYTHRIL, OYENTE, and VANDAL flag the modified example as vulnerable, due to the conservative detection strategies they adopt, as discussed before.

Our solution: We distinguish between *benign* and *vulnerable* reentrancies, *i.e.*, reentrancy as a feature *vs.* a bug. We only consider reentrancy to be vulnerable if it can be leveraged to induce a state-inconsistency (SI). Precisely, if two operations **(a)** operate on the same state variable, **(b)** are reachable from public methods, and **(c)** at-least one is

[1] A recent extension [85] of SEREUM adds support for unconditional reentrancy attacks by tracking data-flow dependencies. However, they only track data-flows from storage variables to the parameters of calls. As a result, even with this extension, SEREUM would fail to detect the attack in Example 1.

```
1  function withdrawBalance(uint amount) public {
2      //[Step 1]: Enter when mutex is false
3      //[Step 4]: Early return, since mutex is true
4      if (mutex == false) {
5        //[Step 2]: mutex = true prevents re-entry
6        mutex = true;
7        if (userBalance[msg.sender] > amount) {
8           //[Step 3]: Attempt to reenter
9           msg.sender.call.value(amount)("");
10          userBalance[msg.sender] -= amount;
11       }
12       mutex = false;
13     }
14 }
15
16 function transfer(address to, uint amt) public {
17     if (mutex == false) {
18         mutex = true;
19         if (userBalance[msg.sender] > amt) {
20             userBalance[to] += amt;
21             userBalance[msg.sender] -= amt;
22         }
23         mutex = false;
24     }
25 }
```

Figure 3.5: Line 6 sets `mutex` to `true`, which prohibits an attacker from reentering by invalidating the path condition (Line 4).

a `write`—we call these two operations a hazardous access pair. The notion of hazardous access unifies both Stale Read (SR), and Destructive Write (DW). SAILFISH performs a lightweight static analysis to detect such hazardous accesses. Since the modified Example 1 (without the `updateSplit`) presented above does not contain any hazardous access pair, we do not flag it as vulnerable.

Scalability. Any SOLIDITY method marked as either `public` or `external` can be called by an external entity *any* number of times in *any* arbitrary order—which translates to an unbounded search space during static reasoning. SECURIFY [25] relies on a `Datalog`-based data-flow analysis, which might fail to reach a fixed point in a reasonable amount of time, as the size of the contract grows. MYTHRIL [27] and OYENTE [28] are symbolic-execution-based tools that share the common problems suffered by any symbolic engine.

Our solution: In SAILFISH, the symbolic verifier validates a program path involving

Table 3.1: Comparison of smart-contract bug-finding tools.

Tool	Cr.	Haz.	Scl.	Off.
SECURIFY [25]	○	○	◐	●
VANDAL [84]	○	○	●	●
MYTHRIL [27]	○	○	○	●
OYENTE [28]	○	○	◐	●
SEREUM [24]	●	○	●	○
SAILFISH	●	●	●	●

Table 3.2: ● Full ◐ Partial ○ No support. **Cr.**: Cross-function, **Haz.**: Hazardous access, **Scl.**: Scalability, **Off.**: Offline detection

hazardous accesses. Unfortunately, the path could access state variables that are likely to be used elsewhere in the contract. It would be very expensive for a symbolic checker to perform a whole-contract analysis required to precisely model those state variables. We augment the verifier with a *value summary* that over-approximates the side-effects of the public methods on the state variables across all executions. This results in an inexpensive symbolic evaluation that conservatively prunes false positives.

Offline bug detection. Once deployed, a contract becomes immutable. Therefore, it is important to be able to detect bugs prior to the deployment. However, offline (static) approaches come with their unique challenges. Unlike an online (dynamic) tool that detects an ongoing attack in just one execution, a static tool needs to reason about all possible combinations of the contract's public methods while analyzing SI issues. As a static approach, SAILFISH needs to tackle all these challenges.

3.2.4 Sailfish overview

This section provides an overview (Figure 3.6) of SAILFISH which consists of the EXPLORER and the REFINER modules.

Explorer. From a contract's source, SAILFISH statically builds a *storage dependency graph*

(SDG) (Section 3.4.1) which over-approximates the read-write accesses (Section 3.4.2) on the storage variables along all possible execution paths. State-inconsistency (SI) vulnerabilities are modeled as graph queries over the SDG. If the query results in an empty set, the contract is certainly non-vulnerable. Otherwise, we generate a counter-example which is subject to further validation by the REFINER.

Example 1 *Example 1 (*Figure 3.4*) contains a reentrancy bug that spans across two functions. The attacker is able to create an SI by leveraging hazardous accesses—`splits[id]` influences (read) the argument of the external call at* Line 16 *in* `splitFunds`*, and it is set (write) at* Line 5 *in* `updateSplit`*. The counter-example returned by the* EXPLORER *is* ⓫ → ⓬ → ⓰ → ❹ → ❺*. Similarly, in Example 2 (*Figure 3.5*), when* `withdrawBalance` *is composed with* `transfer` *to model a cross-function attack,* SAILFISH *detects the write at* Line 10*, and the read at* Line 19 *as hazardous. Corresponding counter-example is* ❹ *...* ❾ → ⓱ *...* ⓳*. In both the cases, the* EXPLORER *detects a potential SI, so conservatively they are flagged as possibly vulnerable. However, this is incorrect for Example 2. Thus, we require an additional step to refine the initial results.*

Refiner. Although the counter-examples obtained from the EXPLORER span across only two public functions $\texttt{P}_1$ and $\texttt{P}_2$, the path conditions in the counter-examples may involve state variables that can be operated on by the public methods $\texttt{P}^*$ other than those two. For example, in case of reentrancy, the attacker can alter the contract state by invoking $\texttt{P}^*$ after the external call-site—which makes reentry to $\texttt{P}_2$ possible. To alleviate this issue, we perform a contract-wide value-summary analysis that computes the necessary pre-conditions to set the values of storage variables. The symbolic verifier consults the value summary when evaluating the path constraints.

Example 2 *In Example 2 (*Figure 3.5*), the* REFINER *would conservatively assume the* `mutex` *to be unconstrained after the external call at* Line 9 *in absence of a value summary –*

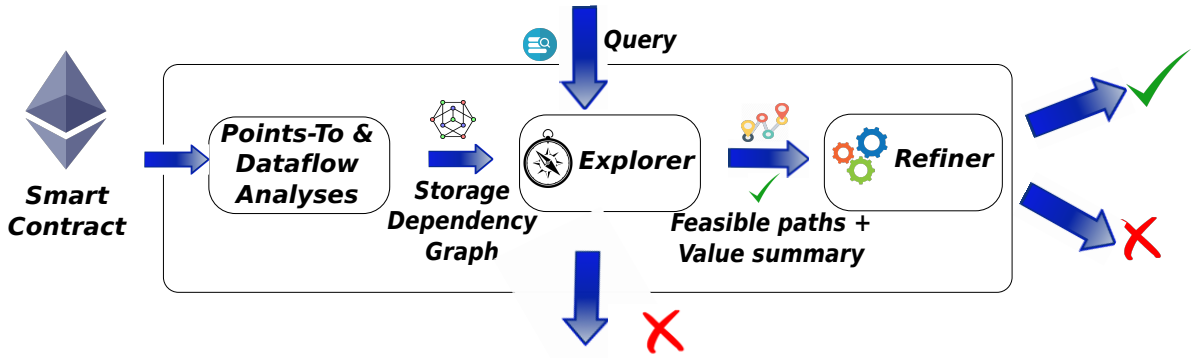

Figure 3.6: Overview of SAILFISH

*which would make the path condition feasible. However, the summary (*Section 3.5*) informs the symbolic checker that all the possible program flows require the* `mutex` *already to be* `false`*, in order to set the* `mutex` *to* `false` *again. Since the pre-condition conflicts with the program-state* $\delta = \{\texttt{mutex} \mapsto \texttt{true}\}$ *(set by* Line 6*), SAILFISH refutes the possibility of the presence of a reentrancy, thereby pruning the false warning.*

3.3 State Inconsistency bugs

In this section, we introduce the notion of state-inconsistency, and how it is related to reentrancy and TOD bugs.

Let $\vec{\mathcal{F}}$ be the list of all public/external functions in a contract $\mathcal{C}$ defined later in Figure 3.11. For each function $\mathcal{F} \in \vec{\mathcal{F}}$, we denote $\mathcal{F}.\mathsf{statements}$ to be the statements of $\mathcal{F}$, and $f = \mathcal{F}.\mathsf{name}$ to be the name of $\mathcal{F}$. In Ethereum, one or more functions can be invoked in a transaction T. Since the contract code is executed by the EVM, the value of its *program counter* (PC) deterministically identifies every statement $s \in \mathcal{F}.\mathsf{statements}$ during run-time. An *event* $e = \langle pc, f(\vec{x}), inv \rangle$ is a 3-tuple that represents the inv-th invocation of the function $\mathcal{F}$ called from outside (*i.e.*, external to the contract $\mathcal{C}$) with arguments $\vec{x}$. Identical invocation of a function $\mathcal{F}$ is associated with the same arguments. For events, we disregard in-

ternal subroutine calls, *e.g.*, if the function $\mathcal{F}$ calls another public function $\mathcal{G}$ from inside its body, the latter invocation does not generate an event. In other words, the notion of events captures the occurrences when a public/external method of a contract is called externally, *i.e.*, across the contract boundary. Functions in events can be called in two ways: either directly by T, or by another contract. If an external call statement $s_c \in \mathcal{F}_c.\mathsf{statements}$ results in a reentrant invocation of $\mathcal{F}$, then *pc* holds the value of the program counter of s_c. In this case, we say that the execution of $\mathcal{F}$ is *contained* within that of $\mathcal{F}_c$. However, the value $pc = 0$ indicates that $\mathcal{F}$ is invoked by T, and not due to the invocation of any other method in $\mathcal{C}$.

Definition 1 *(**Schedule**). A schedule $\mathcal{H} = [e_1, e_2, ..., e_n]$, $\forall e \in \mathcal{H}, e.f \in \{\mathcal{F}.\mathsf{name} | \mathcal{F} \in \vec{\mathcal{F}}\}$ is a valid sequence of n events that can be executed by the EVM. The events, when executed in order on an initial contract state Δ, yield the final state Δ', i.e., $\Delta \xrightarrow{e_1} \Delta_1 \xrightarrow{e_2} \Delta_2 ... \xrightarrow{e_n} \Delta'$, which we denote as $\Delta \xrightarrow{\mathcal{H}} \Delta'$. The set of all possible schedules is denoted by $\mathbb{H}$.*

Definition 2 *(**Equivalent schedules**). Two schedules $\mathcal{H}_1$ and $\mathcal{H}_2$, where $|\mathcal{H}_1| = |\mathcal{H}_2|$, are equivalent, if $\forall e \in \mathcal{H}_1, \exists e' \in \mathcal{H}_2$ such that $e.f = e'.f \wedge e.inv = e'.inv$, and $\forall e' \in \mathcal{H}_2, \exists e \in \mathcal{H}_1$, such that $e'.f = e.f \wedge e'.inv = e.inv$. We denote it by $\mathcal{H}_1 \equiv \mathcal{H}_2$.*

Intuitively, equivalent schedules contain the same set of function invocations.

Definition 3 *(**Transformation function**). A transformation function $\mu : \mathbb{H} \rightarrow \mathbb{H}$ accepts a schedule $\mathcal{H}$, and transforms it to an equivalent schedule $\mathcal{H}' \equiv \mathcal{H}$, by employing one of two possible strategies at a time—**(i)** mutates pc of an event $\exists e' \in \mathcal{H}'$, such that $e'.pc$ holds a valid non-zero value, **(ii)** permutes $\mathcal{H}$. These strategies correspond to two possible ways of transaction ordering, respectively: **(a)** when a contract performs an external call, it can be leveraged to re-enter the contract through internal transactions, **(b)** the external transactions of a contract can be mined in any arbitrary order.*

Definition 4 *(**State inconsistency bug**). For a contract instance $\mathcal{C}$, an initial state Δ, and a schedule $\mathcal{H}_1$ where $\forall e \in \mathcal{H}_1, e.pc = 0$, if there exists a schedule $\mathcal{H}_2 = \mu(\mathcal{H}_1)$,*

where $\Delta \xrightarrow{\mathcal{H}_1} \Delta_1$ *and* $\Delta \xrightarrow{\mathcal{H}_2} \Delta_2$*, then* $\mathcal{C}$ *is said to have a state-inconsistency bug, iff* $\Delta_1 \neq \Delta_2$.

Definition 5 *(**Reentrancy bug**). If a contract* $\mathcal{C}$ *contains an SI bug due to two schedules* $\mathcal{H}_1$ *and* $\mathcal{H}_2 = \mu(\mathcal{H}_1)$*, such that* $\exists e \in \mathcal{H}_2$ $(e.pc \neq 0)$ *(first transformation strategy), then the contract is said to have a reentrancy bug.*

In other words, $e.pc \neq 0$ *implies that* $e.f$ *is a reentrant invocation due to an external call in* $\mathcal{C}$.

Definition 6 *(**Generalized TOD bug**). If a contract* $\mathcal{C}$ *contains an SI bug due to two schedules* $\mathcal{H}_1$ *and* $\mathcal{H}_2 = \mu(\mathcal{H}_1)$*, such that* $\mathcal{H}_2$ *is a permutation (second transformation strategy) of* $\mathcal{H}_1$*, then the contract is said to have a generalized transaction order dependence (G-TOD), or event ordering bug (EO) [86].*

Permutation of events corresponds to the fact that the transactions can be re-ordered due to the inherent non-determinism in the network, e.g., miner's scheduling strategy, gas supplied, etc. In this work, we limit the detection to only those cases where Ether transfer is affected by state-inconsistency—which is in line with the previous work [25, 28]. We refer to those as TOD bugs.

3.4 EXPLORER: Lightweight exploration over SDG

This section introduces the storage dependency graph (SDG), a graph abstraction that captures the control and data flow relations between the storage variables and the critical program instructions, *e.g.*, control-flow deciding, and state-changing operations of a smart contract. To detect SI bugs, we then define hazardous access, which is modeled as queries over the SDG.

3.4.1 Storage dependency graph (SDG)

In a smart contract, the public methods are the entry-points which can be called by an attacker. SAILFISH builds a storage dependency graph (SDG) $\mathcal{N} = (V, E, \chi)$ that models the execution flow as if it was subverted by an attacker, and how the subverted flow impacts the global state of the contract. Specifically, the SDG encodes the following information:

Nodes. A node of an SDG represents either a storage variable, or a statement operating on a storage variable. If $\mathcal{V}$ be the set of all storage variables of a contract, and $\mathcal{S}$ be the statements operating on $\mathcal{V}$, the set of nodes $V := \{\mathcal{V} \cup \mathcal{S}\}$.

Edges. An edge of an SDG represents either the data-flow dependency between a storage variable and a statement, or the relative ordering of statements according to the program control-flow. $\chi(E) \rightarrow \{\texttt{D}, \texttt{W}, \texttt{O}\}$ is a labeling function that maps an edge to one of the three types. A directed edge $\langle u, v \rangle$ from node u to node v is labeled as **(a)** `D`; if $u \in \mathcal{V}, v \in \mathcal{S}$, and the statement v is data-dependent on the state variable u **(b)** `W`; if $u \in \mathcal{S}, v \in \mathcal{V}$, and the state variable v is written by the statement u **(c)** `O`; if $u \in \mathcal{S}, v \in \mathcal{S}$, and statement u precedes statement v in the control-flow graph.

We encode the rules for constructing an SDG in Datalog. First, we introduce the reader to Datalog preliminaries, and then describe the construction rules.

Datalog preliminaries. A Datalog program consists of a set of *rules* and a set of *facts*. Facts simply declare predicates that evaluate to true. For example, `parent("Bill", "Mary")` states that Bill is a parent of Mary. Each Datalog rule defines a predicate as a conjunction of other predicates. For example, the rule: `ancestor(x, y) :- parent(x, z), ancestor(z, y)`—says that `ancestor(x, y)` is true, if both `parent(x, z)` and `ancestor(z, y)` are true. In addition to variables, predicates can also contain constants, which are surrounded by double quotes, or "don't cares", denoted by underscores.

$\mathsf{reach}(s_1, s_2)$	$:-$	s_2 is reachable from s_1
$\mathsf{intermediate}(s_1, s_2, s_3)$	$:-$	$\mathsf{reach}(s_1, s_2), \mathsf{reach}(s_2, s_3)$
$\mathsf{succ}(s_1, s_2)$	$:-$	s_2 is the successor of s_1
$\mathsf{extcall}(s, cv)$	$:-$	s is an external call, cv is the call value
$\mathsf{entry}(s, m)$	$:-$	s is an entry node of method m
$\mathsf{exit}(s, m)$	$:-$	s is an exit node of method m
$\mathsf{storage}(v)$	$:-$	v is a storage variable
$\mathsf{write}(s, v)$	$:-$	s updates variable v
$\mathsf{depend}(s, v)$	$:-$	s is data-flow dependent on v
$\mathsf{owner}(s)$	$:-$	only owner executes s

Figure 3.7: Built-in rules for ICFG related predicates.

Base ICFG facts. The base facts of our inference engine describe the instructions in the application's inter-procedural control-flow graph (ICFG). In particular, Figure 3.7 shows the base rules that are derived from a classical ICFG, where s, m and v correspond to a statement, method, and variable respectively. SAILFISH uses a standard static taint analysis out-of-the-box to restrict the entries in the extcall predicate. Additionally, $\mathsf{owner}(s)$ represents that s can *only* be executed by contract owners, which enables SAILFISH to model SI attacks precisely. In the context of smart contract, the *owner* refers to one or more addresses that play certain administrative roles, *e.g.*, contract creation, destruction, *etc.* Typically, critical functionalities of the contract can only be exercised by the owner. We call the statements that implement such functionalities as *owner-only* statements. Determining the precise set of owner-only statements in a contract can be challenging as it requires reasoning about complex path conditions. SAILFISH, instead, computes a over-approximate set of owner-only statements during the computation of base ICFG facts. This enables SAILFISH, during the EXPLORE phase, not to consider certain hazardous access pairs that can not be exercised by an attacker. To start with, SAILFISH initializes the analysis by collecting the set of storage variables (owner-only variables) $\mathcal{O}$ defined during the contract creation. Then, the algorithm computes the transitive closure of all the storage variables which have *write* operations that are control-flow dependent on

$\mathcal{O}$. Finally, to compute the set of owner-only statements, SAILFISH collects the statements which have their execution dependent on $\mathcal{O}$.

$$
\begin{array}{rcl}
\mathsf{sdg}(s_1, v, \texttt{'W'}) & :- & \mathsf{write}(s_1, v), \mathsf{storage}(v) \\
\mathsf{sdg}(s_1, v, \texttt{'D'}) & :- & \mathsf{depend}(s_1, v), \mathsf{storage}(v) \\
\mathsf{sdg}(s_1, s_2, \texttt{'O'}) & :- & \mathsf{sdg}(s_1, \_, \_), \mathsf{reach}(s_1, s_2), \mathsf{sdg}(s_2, \_, \_), \\
 & & \neg\mathsf{intermediate}(s_1, \_, s_2) \\
\mathsf{sdg}(s_1, s_2, \texttt{'O'}) & :- & \mathsf{extcall}(s_1, \_), \mathsf{entry}(s_2, \_) \\
\mathsf{sdg}(s_4, s_3, \texttt{'O'}) & :- & \mathsf{extcall}(s_1, \_), \mathsf{entry}(\_, m_0), \\
 & & \mathsf{succ}(s_1, s_3), \mathsf{exit}(s_4, m_0)
\end{array}
$$

Figure 3.8: Rules for constructing SDG.

SDG construction. The basic facts generated from the previous step can be leveraged to construct the SDG. As shown in Fig 3.8, a "write-edge" of an SDG is labeled as `'W'`, and is constructed by checking whether storage variable v gets updated in statement s. Similarly, a "data-dependency edge" is labeled as `'D'`, and is constructed by determining whether the statement s is data-dependent on the storage variable v. Furthermore, we also have the "order-edge" to denote the order between two statements, and those edges can be drawn by checking the reachability between nodes in the original ICFG. Finally, an external call in SOLIDITY can be weaponized by the attacker by hijacking the current execution. In particular, once an external call is invoked, it may trigger the callback function of the attacker who can perform arbitrary operations to manipulate the storage states of the original contract. To model these semantics, we also add extra `'O'`-edges to connect external calls with other public functions that can potentially update storage variables that may influence the execution of the current external call. Specifically, we add an extra order-edge to connect the external call to the entry point of another public function m, as well as an order-edge from the exit node of m to the successor of the original external call.

Example 3 *Consider Example 1 (*Figure 3.4*) that demonstrates an SI vulnerability due*

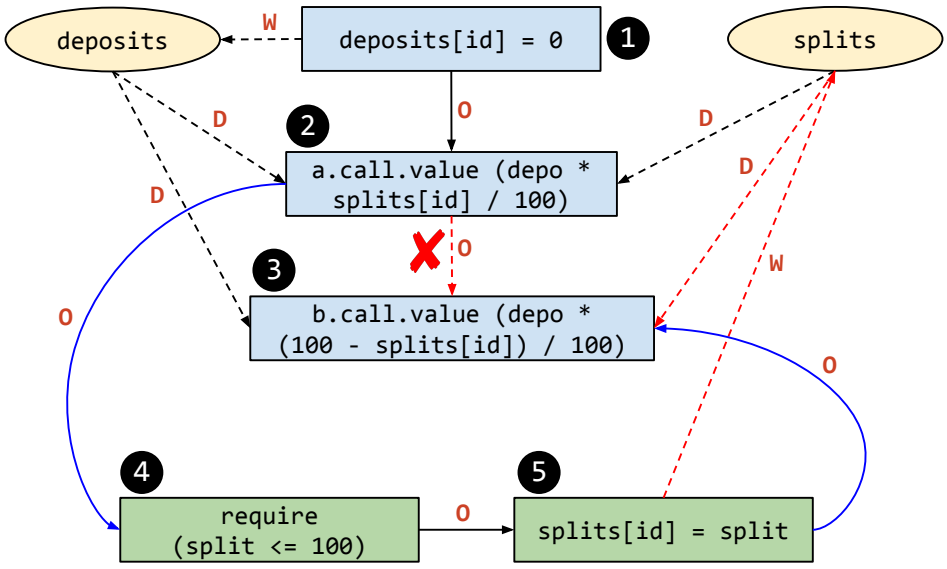

Figure 3.9: SDG for Example 1. Ovals and rectangles represent storage variables and instructions. Blue [ ] and green [ ] colored nodes correspond to instructions from `splitFunds` and `updateSplit` methods, respectively. The `O`, `D`, and `W` edges stand for `order`, `data`, and `write` edges, respectively. The red [ ] edges on `splits` denote hazardous access.

to both ***splitFunds*** *and* ***updateSplit*** *methods operating on a state variable* ***splits[id]***. Figure 3.9 *models this attack semantics.* ***deposits*** *and* ***splits[id]*** *correspond to the variable nodes in the graph.* Line 12 *writes to* ***deposits****; thus establishing a* `W` *relation from the instruction to the variable node.* Line 16 *and* Line 19 *are data-dependent on both the state variables. Hence, we connect the related nodes with* `D` *edges. Finally, the instruction nodes are linked together with directed* `O` *edges following the control-flow. To model the reentrancy attack, we created an edge from the external call node* ❷ → ❹*, the entry point of* ***splitFunds****. Next, we remove the edge between the external call* ❷*, and its successor* ❸*. Lastly, we add an edge between* ❺*, the exit node of* ***updateSplit****, and* ❸*, the following instruction in* ***updateSplit****.*

3.4.2 Hazardous access

Following our discussion in Section 3.3, to detect SI bugs in a smart contract, one needs to enumerate and evaluate all possible schedules on every contract state—which is computationally infeasible. To enable scalable detection of SI bugs statically, we define *hazardous access*, which is inspired by the classical data race problem, where two different execution paths operate on the same storage variable, and at least one operation is a *write*. In a smart contract, the *execution paths* correspond to two executions of public function(s).

As shown in the hazard(.) predicate in Figure 3.10, a hazardous access is a tuple denoted by $\langle s_1, s_2, v \rangle$, where v is a storage variable which both the statements s_1 and s_2 operate on, and either s_1, or s_2, or both are *write* operations. While deriving the data-flow dependency predicate $\mathsf{sdg}(s, v, \text{'D'})$, we consider both direct and indirect dependencies of the variable v. We say that a statement s operates on a variable v if either s is an assignment of variable v or s contains an expression that is dependent on variable s.

SAILFISH identifies hazardous access statically by querying the contract's SDG, which is a path-condition agnostic data structure. A non-empty query result indicates the existence of a hazardous access. However, these accesses might not be feasible in reality due to conflicting path conditions. The REFINER module (Section 3.5) uses symbolic evaluation to prune such infeasible accesses.

3.4.3 State inconsistency bug detection

As discussed in Section 3.3, a smart contract contains an SI bug if there exists two schedules that result in a different contract state, *i.e.*, the values of the storage variables. Instead of enumerating all possible schedules (per definition) statically which is computationally infeasible, we use hazardous access as a *proxy* to detect the root cause of SI. Two schedules can result in different contract states if: **(a)** there exist two operations,

where at least one is a *write* access, on a common storage variable, and **(b)** the relative order of such operations differ in two schedules. The hazardous access captures the first (**a**) condition. Now, in addition to hazardous access, SI bugs require to hold certain conditions that can alter (**b**) the relative order of the operations in the hazardous access pair. For reentrancy, SAILFISH checks if a hazardous access pair is reachable in a reentrant execution, as it can alter the execution order of the statements in a hazardous access pair. To detect TOD, SAILFISH checks whether an Ether transfer call is reachable from one of the statements in a hazardous access pair. In this case, the relative execution order of those statements determines the amount of Ether transfer.

Reentrancy detection. A malicious reentrancy query (Figure 3.10) looks for a hazardous access pair $\langle s_1, s_2 \rangle$ such that both s_1 and s_2 are reachable from an external call in the SDG, and executable by an attacker.

To detect *delegate-based* reentrancy attacks, where the delegatecall destination is tainted, we treat delegatecall in the same way as the extcall in Figure 3.10. For untainted delegatecall destinations, if the source code of the delegated contract is available, SAILFISH constructs an SDG that combines both the contracts. If neither the source, nor the address of the delegated contract is available, SAILFISH treats delegatecall in the same way as an unsafe external call. For *create-based* attacks, since the source code of the child contract is a part of the parent contract, SAILFISH builds the SDG by combining both the creator (parent) and the created (child) contracts. Subsequently, SAILFISH leverages the existing queries in Figure 3.10 on the combined SDG. For untainted extcall, and delegatecall destinations, SAILFISH performs inter-contract analysis to build an SDG combining both contracts. To model inter-contract interaction as precisely as possible, we perform a backward data-flow analysis starting from the destination d of an external call (*e.g.*, call, delegatecall, *etc.*), which leads to the following three possibilities: **(a)** d is visible from source, **(b)** d is set by the *owner* at run-time, *e.g.*, in the constructor during contract

$$
\begin{aligned}
\mathsf{hazard}(s_1, s_2, v) \ &:- \ \mathsf{storage}(v), \mathsf{sdg}(s_1, v, \texttt{'W'}),\\
&\quad\ \mathsf{sdg}(s_2, v, \_), s_1 \neq s_2\\
\mathsf{reentry}(s_1, s_2) \ &:- \ \mathsf{extcall}(e, \_), \mathsf{reach}(e, s_1), \mathsf{reach}(e, s_2),\\
&\quad\ \mathsf{hazard}(s_1, s_2, \_), \neg\mathsf{owner}(s_1), \neg\mathsf{owner}(s_2)\\
\mathsf{tod}(s_1, s_2) \ &:- \ \mathsf{extcall}(e, cv), cv > 0, \mathsf{reach}(s_1, e),\\
&\quad\ \mathsf{hazard}(s_1, s_2, \_), \neg\mathsf{owner}(s^*),\\
&\quad\ s^\star \in \{s_1, s_2\}\\
\text{Base case :}&\\
\mathsf{cex}(s_0, s_1) \ &:- \ \mathsf{entry}(s_0, \_), \mathsf{succ}(s_0, s_1), \mathsf{f}(s_1, s_2),\\
&\quad\ \mathsf{extcall}(s', \_), \mathsf{reach}(s_1, s^\star),\\
&\quad\ s^\star \in \{s_1, s_2, s'\}, f \in \{\mathsf{tod}, \mathsf{reentry}\}\\
\text{Inductive case :}&\\
\mathsf{cex}(s_1, s_2) \ &:- \ \mathsf{cex}(\_, s_1), \mathsf{succ}(s_1, s_2), \mathsf{f}(s_3, s_4),\\
&\quad\ \mathsf{extcall}(s', \_), \mathsf{reach}(s_2, s^\star),\\
&\quad\ s^\star \in \{s_3, s_4, s'\}, f \in \{\mathsf{tod}, \mathsf{reentry}\}
\end{aligned}
$$

Figure 3.10: Rules for hazardous access and counter-examples.

creation. In this case, we further infer d by analyzing existing transactions, *e.g.*, by looking into the arguments of the contract-creating transaction, and **(c)** d is attacker-controlled. While crawling, we build a database from the contract address to its respective source. Hence, for cases (a) and (b) where d is statically known, we incorporate the target contract in our analysis if its source is present in our database. If either the source is not present, or d is tainted (case (c)), we treat such calls as *untrusted*, requiring no further analysis.

Example 4 *When run on the SDG in* Figure 3.9 *(Example 1), the query returns the tuple* $\langle 3, 5\rangle$*, because they both operate on the state variable* `splits`*, and belong to distinct public methods, viz.,* `splitFunds` *and* `updateSplit` *respectively.*

TOD detection. As explained in Section 3.1, TOD happens when Ether transfer is affected by re-ordering transactions. Hence, a hazardous pair $\langle s_1, s_2\rangle$ forms a TOD if the following conditions hold: 1) an external call is reachable from either s_1 or s_2, and 2) the amount of Ether sent by the external call is greater than zero.

SAILFISH supports all three TOD patterns supported by SECURIFY [25]—**(i)** `TOD`

`Transfer` specifies that the pre-condition of an Ether transfer, *e.g.*, a condition c guarding the transfer, is influenced by transaction ordering, **(ii)** `TOD Amount` indicates that the amount a of Ether transfer is dependent on transaction ordering, and **(iii)** `TOD Receiver` defines that the external call destination e is influenced by the transaction ordering. To detect these attacks, SAILFISH reasons if c, or a, or e is data-flow dependent on some $\mathsf{storage}(v)$, and the statements corresponding to those three are involved in forming a hazardous pair.

Counter-example generation. If a query over the SDG returns $\perp$ (empty), then the contract is safe, because the SDG models the state inconsistency in the contract. On the other hand, if the query returns a list of pairs $\langle s_1, s_2 \rangle$, SAILFISH performs a *refinement* step to determine if those pairs are indeed feasible. Since the original output pairs (*i.e.*, $\langle s_1, s_2 \rangle$) can not be directly consumed by the symbolic execution engine, SAILFISH leverages the `cex`-rule in Figure 3.10 to compute the minimum ICFG G that contains statements s_1, s_2, and the relevant external call s'. In the base case, `cex`-rule includes edges between entry points and their successors that can transitively reach s_1, s_2, or s'. In the inductive case, for every node s_1 that is already in the graph, we recursively include its successors that can also reach s_1, s_2, or s'.

Example 5 SAILFISH *extracts the graph slice starting from the root (not shown in* Figure 3.9*) of the SDG to node* ❺*. The algorithm extracts the sub-graph* $\langle \texttt{root} \rangle \xrightarrow{*}$ ❷ $\rightarrow$ ❹ $\rightarrow$ ❺ $\rightarrow$ ❸*, maps all the SDG nodes to the corresponding ICFG nodes, and computes the final path slice which the* REFINER *runs on.*

3.5 REFINER: Symbolic evaluation with value summary

As explained in Section 3.4, if the EXPLORER module reports an alarm, then there are two possibilities: either the contract is indeed vulnerable, or the current counter-example (*i.e.*, subgraph generated by the rules in Figure 3.10) is infeasible. Thus, SAILFISH proceeds to refine the subgraph by leveraging symbolic evaluation (Section 3.5.2). However, as we show later in the evaluation, a naive symbolic evaluation whose storage variables are completely unconstrained will raise several false positives. To address this challenge, the REFINER module in SAILFISH leverages a light-weight *value summary analysis* (Section 3.5.1) that output the potential symbolic values of each storage variable under different constraints, which will be used as the pre-condition of the symbolic evaluation (Section 3.5.2).

3.5.1 Value summary analysis (VSA)

For each storage variable, the goal of value summary analysis (VSA) is to compute its invariant that holds through the life-cycle of a smart contract. While summary-based analysis has been applied in many different applications before, there is no off-the-shelf VSA for smart contracts that we could leverage for the following reasons: **(a) Precision.** A value summary based on abstract interpretation [87] that soundly computes the interval for each storage variable scales well, but since it ignores the path conditions under which the interval holds, it may lead to *weaker preconditions* that are not sufficient to prune infeasible paths. For the example in Figure 3.5, a naive and scalable analysis will ignore the control flows, and conclude that the summary of `mutex` is $\top$ (either `true` or `false`), which will be useless to the following symbolic evaluation, since `mutex` is unconstrained.

(b) Scalability. A *path-by-path* summary [80, 81] that relies on symbolic execution first computes the pre-condition pre_w, post-condition $post_w$, and per-path summary $\phi_w = pre_w \wedge post_w$ for every path w. The overall summary ϕ_f of the function f is the disjunction of individual path summaries, *i.e.*, $\phi_f = \vee_w \phi_w$. We identify the following barriers in adopting this approach out of the box: **(i)** *Generation*: The approach is computationally intensive due to well-known path explosion problem. **(ii)** *Application*: The summary being the unification of the constraints collected along all the paths, such a summary is complex, which poses a significant challenge to the solver. In fact, when we evaluated our technique by plugging in a similar path-by-path summary, the analysis timed out for 21.50% of the contracts due to the increased cost of the REFINE phase. **(iii)** *Usability*: Lastly, such a summary is precise, yet expensive. Computing a precise summary is beneficial only when it is used sufficient times. Our aim is to build a usable system that scales well in two dimensions—both to large contracts, and a large number of contracts. As the dataset is deduplicated, the scope of reusability is narrow. Therefore, an expensive summary does not pay off well given our use case. What we need in SAILFISH is a summarization technique that has a small resource footprint, yet offers reasonable precision for the specific problem domain, *i.e.*, smart contracts.

Therefore, we design a domain-specific VSA (Figure 3.12) to tackle both the challenges: **(a) Precision**: Unlike previous scalable summary techniques that map each variable to an interval whose path conditions are merged, we compensate for such precision loss at the merge points of the control flows using an idea inspired by symbolic union [88]—our analysis stitches the branch conditions to their corresponding symbolic variables at the merge points. **(b) Scalability**: **(i)** *Generation*: This design choice, while being more precise, could still suffer from path explosion. To mitigate this issue, our analysis first starts with a precise abstract domain that captures concrete values and their corresponding path conditions, and then *gradually sacrifices* the precision in the context of statements

$$
\begin{array}{rcl}
\text{Program } \mathcal{P} & ::= & (\delta, \pi, \vec{\mathcal{F}}) \\
\text{ValueEnv } \delta & ::= & V \rightarrow \text{Expr} \\
\text{PathEnv } \pi & ::= & \mathsf{loc} \rightarrow C \\
\text{Expr } e & ::= & x \mid c \mid op(\vec{e}) \mid S(\vec{e}) \\
\text{Statement } s & ::= & \mathsf{havoc}(s) \mid l := e \mid s; s \mid r = f(\vec{e}) \\
& \mid & (\texttt{if } e\ s\ s) \mid (\texttt{while } e\ s) \\
\text{Function } \mathcal{F} & ::= & \textbf{function } f(\vec{x})\ s \textbf{ returns } y
\end{array}
$$

$$x, y \in \textbf{Variable} \quad c \in \textbf{Constant} \quad S \in \textbf{StructName}$$

Figure 3.11: Syntax of our simplified language.

that are difficult, or expensive to reason about, *e.g.*, loops, return values of external calls, updates over nested data structures, *etc.* **(ii)** *Application*: Lastly, we carefully design the evaluation rules (If-rule in Figure 3.12) that selectively drop path conditions at the confluence points—which leads to simpler constraints at the cost of potential precision loss. However, our evaluation of SAILFISH suggests that, indeed, our design of VSA strikes a reasonable trade-off in the precision-scalability spectrum in terms of both bug detection and analysis time.

To formalize our rules for VSA, we introduce a simplified language in Figure 3.11. In particular, a contract $\mathcal{P}$ consists of **(a)** a list of public functions $\vec{\mathcal{F}}$ (private functions are inline), **(b)** a value environment δ that maps variables or program identifiers to concrete or symbolic values, and **(c)** a path environment π that maps a location *loc* to its path constraint C. It is a boolean value encoding the branch decisions taken to reach the current state. Moreover, each function $\mathcal{F}$ consists of arguments, return values, and a list of statements containing loops, branches, and sequential statements, *etc.* Our expressions e include common features in SOLIDITY such as storage access, struct initialization, and arithmetic expressions (function invocation is handled within a statement), *etc.* Furthermore, since all private functions are inline, we assume that the syntax for calling an external function with return variable r is $r = f(\vec{e})$. Finally, we introduce a havoc operator to make those variables in hard-to-analyze statements

unconstrained, *e.g.*, havoc(s) changes each variable in s to $\top$ (completely unconstrained).

Figure 3.12 shows a representative subset of the inference rules for computing the summary. A program state consists of the value environment δ and the path condition π. A rule $\langle e, \delta, \pi\rangle \rightsquigarrow \langle v, \delta', \pi'\rangle$ says that a successful execution of e in the program state $\langle \delta, \pi\rangle$ results in value v and the state $\langle \delta', \pi'\rangle$.

Bootstrapping. The value summary procedure starts with the "contract" rule that sequentially generates the value summary for each public function $\mathcal{F}_i$ (all non-public methods are inline). The output value environment δ' contains the value summary for all storage variables. More precisely, for each storage variable s, δ' maps it to a set of pairs $\langle \pi, v\rangle$ where v is the value of s under the constraint π. Similarly, to generate the value summary for each function $\mathcal{F}_i$, SAILFISH applies the "Func" rule to visit every statement s_i inside method $\mathcal{F}_i$.

Expression. There are several rules to compute the rules for different expressions e. In particular, if e is a constant c, the value summary for e is c itself. If e is an argument of a public function $\mathcal{F}_i$ whose values are completely under the control of an attacker, the "Argument" rule will havoc e and assume that its value can be any value of a particular type.

Helper functions. The `dom`(δ) returns all the keys of an environment δ. The `lhs`(e) returns variables written by e.

Collections. For a variable of type Array or Map, our value summary rules do not differentiate elements under different indices or keys. In particular, for a variable a of type array, the "store" rule performs a weak update by unioning all the previous values stored in a with the new value e_0. We omit the rule for the map since it is similar to an array. Though the rule is imprecise as it loses track of the values under different indices, it summarizes possible values that are stored in a.

Assignment. The "assign" rule essentially keeps the value summaries for all variables

from the old value environment δ except for mapping e_0 to its new value e_1.

External calls. Since all private and internal functions are assumed to be inline, we assume all function invocations are external. As we do not know how the attacker is going to interact with the contract via external calls, we assume that it can return arbitrary values. Here is the key intuition of the "ext" rule: for any invocation to an external function, we havoc its return variable r.

Loop. Finally, since computing value summaries for variables inside loop bodies are very expensive and hard to scale to complex contracts, our "loop" rule simply havocs all variables that are written in the loop bodies.

Conditional. Rule "if" employs a meta-function μ to merge states from alternative execution paths.

$$\mu(b, v_1, v_2) = \begin{cases} \{\langle \top, v_1 \rangle\} & \text{if } b == \texttt{true} \\ \{\langle \top, v_2 \rangle\} & \text{if } b == \texttt{false} \\ \{\langle b, v_1 \rangle, \langle \neg b, v_2 \rangle\} & \text{Otherwise} \end{cases}$$

In particular, the rule first computes the symbolic expression v_0 for the branch condition e_0. If v_0 is evaluated to `true`, then the rule continues with the `then` branch e_1 and computes its value summary v_1. Otherwise, the rule goes with the `else` branch e_2 and obtains its value summary v_2. Finally, if the branch condition e_0 is a symbolic variable whose concrete value cannot be determined, then our value summary will include both v_1 and v_2 together with their path conditions. Note that in all cases, the path environment π' needs to be computed by conjoining the original π with the corresponding path conditions that are taken by different branches.

$$\frac{\mathcal{P} = (\delta, \pi, \vec{\mathcal{F}}),\ \langle \mathcal{F}_0, \delta, \pi \rangle \rightsquigarrow \langle \texttt{void}, \delta_1, \pi_1 \rangle \quad \cdots \quad \langle \mathcal{F}_n, \delta_n, \pi_n \rangle \rightsquigarrow \langle \texttt{void}, \delta', \pi' \rangle}{\langle \mathcal{P}, \delta, \pi \rangle \rightsquigarrow \langle \texttt{void}, \delta', \pi' \rangle} \text{ (Contract)}$$

$$\frac{\langle s, \delta, \pi \rangle \rightsquigarrow \langle \texttt{void}, \delta', \pi' \rangle}{\langle (\texttt{function } f(\vec{x}) \ s \ \texttt{returns } y), \delta, \pi \rangle \rightsquigarrow \langle \texttt{void}, \delta', \pi' \rangle} \text{ (Func)}$$

$$\frac{}{\langle c, \delta, \pi \rangle \rightsquigarrow \langle c, \delta, \pi \rangle} \text{ (Const)} \quad \frac{\texttt{isArgument}(a)\ v = \texttt{havoc}(a)}{\langle a, \delta, \pi \rangle \rightsquigarrow \langle v, \delta', \pi \rangle} \text{ (Argument)}$$

$$\frac{\langle e_1, \delta, \pi \rangle \rightsquigarrow \langle v_1, \delta, \pi \rangle \quad \oplus \in \{+, -, *, /\} \quad \langle e_2, \delta, \pi \rangle \rightsquigarrow \langle v_2, \delta, \pi \rangle \quad v = v_1 \oplus v_2}{\langle (e_1 \oplus e_2), \delta, \pi \rangle \rightsquigarrow \langle v, \delta, \pi \rangle} \text{ (Binop)}$$

$$\frac{\langle e_0, \delta, \pi \rangle \rightsquigarrow \langle v_0, \delta, \pi \rangle \quad \delta' = \{y \mapsto \delta(y) \mid y \in \texttt{dom}(\delta) \wedge y \neq a\} \ \cup \ \{a[0] \mapsto (\delta(a[0]) \cup \langle \pi, v_0 \rangle)\}}{\langle (a[i] = e_0), \delta, \pi \rangle \rightsquigarrow \langle \texttt{void}, \delta', \pi \rangle} \text{ (Store)}$$

$$\frac{\langle \_, v \rangle = \delta(a[0])}{\langle a[i], \delta, \pi \rangle \rightsquigarrow \langle v, \delta, \pi \rangle} \text{ (Load)}$$

$$\frac{\delta' = \{y \mapsto \delta(y) \mid y \in \texttt{dom}(\delta) \wedge y \neq e_0\} \ \cup \ \{e_0 \mapsto \langle \pi, e_1 \rangle \cup \delta(e_0)\}}{\langle (e_0 = e_1), \delta, \pi \rangle \rightsquigarrow \langle \texttt{void}, \delta', \pi \rangle} \text{ (Assign)}$$

$$\frac{\delta' = \{y \mapsto \delta(y) \mid y \in \texttt{dom}(\delta) \wedge y \neq r\} \ \cup \ \{r \mapsto \langle \pi, \texttt{havoc}(r) \rangle\}}{\langle r = \texttt{f}(\vec{e}), \delta, \pi \rangle \rightsquigarrow \langle \texttt{void}, \delta', \pi \rangle} \text{ (Ext)}$$

$$\frac{\langle e_0, \delta, \pi \rangle \rightsquigarrow \langle v_0, \delta, \pi \rangle \quad \pi' = \pi \wedge v_0 \quad \delta' = \{y \mapsto \delta(y) \mid y \notin \texttt{lhs}(e_1)\} \ \cup \ \{y \mapsto \langle \pi', \texttt{havoc}(y) \rangle \mid y \in \texttt{lhs}(e_1)\}}{\langle (\texttt{while } e_0 \ e_1), \delta, \pi \rangle \rightsquigarrow \langle v_0, \delta', \pi \wedge \neg v_0 \rangle} \text{ (Loop)}$$

$$\frac{\langle e_0, \delta, \pi \rangle \rightsquigarrow \langle v_0, \delta, \pi \rangle \quad b = isTrue(v_0) \quad \langle e_1, \delta, \pi \wedge b \rangle \rightsquigarrow \langle v_1, \delta_1, \pi_1 \rangle \quad \langle e_2, \delta, \pi \wedge \neg b \rangle \rightsquigarrow \langle v_2, \delta_2, \pi_2 \rangle \quad \delta' = \delta \cup \delta_1 \cup \delta_2}{\langle (\texttt{if } e_0 \ e_1 \ e_2), \delta, \pi \rangle \rightsquigarrow \langle \mu(b, v_1, v_2), \delta', \pi \rangle} \text{ (If)}$$

Figure 3.12: Inference rules for value summary analysis.

3.5.2 Symbolic evaluation

Based on the rules in Figure 3.10, if the contract contains a pair of statements $\langle s_1, s_2 \rangle$ that match our state-inconsistency query (*e.g.*, reentrancy), the EXPLORER module (Section 3.4) returns a subgraph G (of the original ICFG) that contains statement s_1 and

s_2. In that sense, checking whether the contract indeed contains the state-inconsistency bug boils down to a standard reachability problem in G: does there exist a valid path π that satisfies the following conditions: 1) π starts from an entry point v_0 of a public method, and 2) following π will visit s_1 and s_2, sequentially. [2] Due to the over-approximated nature of our SDG that ignores all path conditions, a valid path in SDG does not always map to a *feasible execution path* in the original ICFG. As a result, we have to symbolically evaluate G and confirm whether π is indeed feasible.

A naive symbolic evaluation strategy is to evaluate G by precisely following its control flows while assuming that all storage variables are completely unconstrained ($\top$). With this assumption, as our ablation study shows (Figure 3.18), SAILFISH fails to refute a significant amount of false alarms. So, the key question that we need to address is: How can we symbolically check the reachability of G while constraining the range of storage variables without losing too much precision? This is where VSA comes into play. Recall that the output of our VSA maps each storage variable into a set of abstract values together with their corresponding path constraints in which the values hold. Before invoking the symbolic evaluation engine, we union those value summaries into a global pre-condition that is enforced through the whole symbolic evaluation.

Example 6 *Recall in Fig 3.5, the* EXPLORER *reports a false alarm due to the over-approximation of the SDG. We now illustrate how to leverage VSA to refute this false alarm.*

Step 1: *By applying the VSA rules in* Figure 3.12 *to the contract in* Figure 3.5*,* SAILFISH *generates the summary for storage variable* ***mutex****:* $\{\langle \mathsf{mutex} = \mathsf{false}, \mathtt{false}\rangle, \langle \mathsf{mutex} = \mathsf{false}, \mathtt{true}\rangle\}$. *In other words, after invoking any sequence of public functions,*

[2] Since TOD transfer requires reasoning about two different executions of the same code, we adjust the goal of symbolic execution for TOD as the following: Symbolic evaluate subgraph G twice (one uses *true* as pre-condition and another uses value summary). The amount of Ether in the external call are denoted as a_1, a_2, respectively. We report a TOD if $a_1 \neq a_2$.

`mutex` can be updated to `true` or `false`, if pre-condition `mutex==false` holds. Here, we omit the summary of other storage variables (e.g., `userBalance`) for simplicity.

Step 2: *Now, by applying the symbolic checker on the `withdrawBalance` function for the first time,* SAILFISH *generates the following path condition* π*:* `mutex` $==$ `false` $\wedge$ `userBalance`[`msg.sender`] $>$ `amount` *as well as the following program state* δ *before invoking the external call at* Line 9*:* $\delta = \{$`mutex` $\mapsto$ `true`, ...$\}$

Step 3: *After Step 2, the current program state* δ *indicates that the value of `mutex` is `true`. Note that to execute the `then-branch` of `withdrawBalance`, `mutex` must be `false`. Based on the value summary of `mutex` in Step 1, the pre-condition to set `mutex` to `false` is* `mutex` $=$ `false`*. However, the pre-condition is not satisfiable under the current state* δ*. Therefore, although the attacker can re-enter the `withdrawBalance` method through the callback mechanism, it is impossible for the attacker to re-enter the `then-branch` at* Line 6*, and trigger the external call at* Line 9*. Thus,* SAILFISH *discards the reentrancy report as false positive.*

3.6 Implementation

Explorer. It is a lightweight static analysis that lifts the smart contract to an SDG. The analysis is built on top of the SLITHER [89] framework that lifts SOLIDITY source code to its intermediate representation called SLITHIR. SAILFISH uses SLITHER's API, including the taint analysis, out of the box.

Refiner. SAILFISH leverages ROSETTE [88] to symbolically check the feasibility of the counter-examples. ROSETTE provides support for symbolic evaluation. ROSETTE programs use assertions and symbolic values to formulate queries about program behavior, which are then solved with off-the-shelf SMT solvers. SAILFISH uses `(solve expr)` query that searches for a binding of symbolic variables to concrete values that satisfies the

assertions encountered during the symbolic evaluation of the program expression `expr`.

3.7 Evaluation

In this section, we describe a series of experiments that are designed to answer the following research questions: **RQ1.** How effective is SAILFISH compared to the existing smart contracts analyzers with respect to vulnerability detection? **RQ2.** How scalable is SAILFISH compared to the existing smart contracts analyzers? **RQ3.** How effective is the REFINE phase in pruning false alarms?

3.7.1 Experimental setup

Dataset. We have crawled the source code of all 91,921 contracts from Etherscan [11], which cover a period until October 31, 2020. We excluded 2,068 contracts that either require very old versions (<0.3.x) of the SOLIDITY compiler, or were developed using the VYPER framework. As a result, after deduplication, our evaluation dataset consists of 89,853 SOLIDITY smart contracts. Further, to gain a better understanding of how each tool scales as the size of the contract increases, we have divided the entire dataset, which we refer to as **full** dataset, into three mutually-exclusive sub-datasets based on the number of lines of source code—**small** ($[0, 500)$), **medium** ($[500, 1000)$), and **large** ($[1000, \infty)$) datasets consisting of 73,433, 11,730, and 4,690 contracts, respectively. We report performance metrics individually for all three datasets.

Analysis setup. We ran our analysis on a Celery v4.4.4 [90] cluster consisting of six identical machines running Ubuntu 18.04.3 Server, each equipped with Intel(R) Xeon(R) CPU E5-2690 v2@3.00 GHz processor (40 core) and 256 GB memory.

Analysis of real-world contracts. We evaluated SAILFISH against four other static analysis tools, *viz.*, SECURIFY [25], VANDAL [84], MYTHRIL [27], OYENTE [28], and one

dynamic analysis tool, *viz.*, SEREUM [24]—capable of finding either reentrancy, or TOD, or both. Given the influx of smart contract related research in recent years, we have carefully chosen a representative subset of the available tools that employ a broad range of minimally overlapping techniques for bug detection. SMARTCHECK [91] and SLITHER [89] were omitted because their reentrancy detection patterns are identical to SECURIFY's NW (No Write After Ext. Call) signature.

We run all the static analysis tools, including SAILFISH, on the full dataset under the analysis configuration detailed earlier. If a tool supports both reentrancy and TOD bug types, it was configured to detect both. We summarize the results of the analyses in Table 3.3. For each of the analysis tools and analyzed contracts, we record one of the four possible outcomes– **(a)** *safe*: no vulnerability was detected **(b)** *unsafe*: a potential state-inconsistency bug was detected **(c)** *timeout*: the analysis failed to converge within the time budget (20 minutes) **(d)** *error*: the analysis aborted due to infrastructure issues, *e.g.*, unsupported SOLIDITY version, or a framework bug, *etc.* For example, the latest SOLIDITY version at the time of writing is 0.8.3, while OYENTE supports only up to version 0.4.19.

3.7.2 Vulnerability detection

In this section, we report the fraction (%) of *safe*, *unsafe* (warnings), and timed-out contracts reported by each tool with respect to the total number of contracts successfully analyzed by that tool, excluding the "error" cases.

Comparison against other tools. SECURIFY, MYTHRIL, OYENTE, VANDAL, and SAILFISH report potential reentrancy in 7.10%, 4.18%, 0.99%, 52.27%, and 2.40% of the contracts. Though all five static analysis tools detect reentrancy bugs, TOD detection is supported by only three tools, *i.e.*, SECURIFY, OYENTE, and SAILFISH which raise

Bug	Tool	Safe	Unsafe	Timeout	Error
Reentrancy	SECURIFY	72,149	6,321	10,581	802
	VANDAL	40,607	45,971	1,373	1,902
	MYTHRIL	25,705	3,708	59,296	1,144
	OYENTE	26,924	269	0	62,660
	SAILFISH	83,171	2,076	1,211	3,395
TOD	SECURIFY	59,439	19,031	10,581	802
	OYENTE	23,721	3,472	0	62,660
	SAILFISH	77,692	7,555	1,211	3,395

Table 3.3: Comparison of bug finding abilities of tools

potential TOD warnings in 21.37%, 12.77%, and 8.74% of the contracts.

MYTHRIL, being a symbolic execution based tool, demonstrates obvious scalability issues: It timed out for 66.84% of the contracts. Though OYENTE is based on symbolic execution as well, it is difficult to properly assess its scalability. The reason is that OYENTE failed to analyze most of the contracts in our dataset due to the unsupported SOLIDITY version, which explains the low rate of warnings that OYENTE emits. Unlike symbolic execution, static analysis seems to scale well. SECURIFY timed-out for only 11.88% of the contracts, which is significantly lower than that of MYTHRIL. When we investigated the reason for SECURIFY timing out, it appeared that the `Datalog`-based data-flow analysis (that SECURIFY relies on) fails to reach a fixed-point for larger contracts. VANDAL's static analysis is inexpensive and shows good scalability, but suffers from poor precision. In fact, VANDAL flags as many as 52.27% of all contracts as vulnerable to reentrancy–which makes VANDAL reports hard to triage due to the overwhelming amount of warnings. VANDAL timed out for the least (1.56%) number of contracts. Interestingly, SECURIFY generates fewer reentrancy warnings than MYTHRIL. This can be attributed to the fact that the `NW` policy of SECURIFY considers a write after an external call as vulnerable, while MYTHRIL conservatively warns about both read and write. However, SAILFISH strikes a balance between both scalability and precision as it timed-out only

Tool	Reentrancy			TOD		
	TP	FP	FN	TP	FP	FN
SECURIFY	9	163	17	102	244	8
VANDAL	26	626	0	–	–	–
MYTHRIL	7	334	19	–	–	–
OYENTE	8	16	18	71	116	39
SAILFISH	26	11	0	110	59	0

Table 3.4: Manual determination of the ground truth

for 1.40% of the contracts, and generates the fewest alarms.

Ground truth determination. To be able to provide better insights into the results, we performed manual analysis on a randomly sampled subset of 750 contracts ranging up to 3,000 lines of code, out of a total of 6,581 contracts successfully analyzed by all five static analysis tools, without any timeout or error. We believe that the size of the dataset is in line with prior work [92, 93]. We prepared the ground truth by manually inspecting the contracts for reentrancy and TOD bugs using the following criteria: **(a)** *Reentrancy:* The untrusted external call allows the attacker to re-enter the contract, which makes it possible to operate on an inconsistent internal state. **(b)** *TOD:* A front-running transaction can divert the control-flow, or alter the Ether-flow, *e.g.*, Ether amount, call destination, *etc.*, of a previously scheduled transaction.

In the end, the manual analysis identified 26 and 110 contracts with reentrancy and TOD vulnerabilities, respectively. We then ran each tool on this dataset, and report the number of correct (TP), incorrect (FP), and missed (FN) detection by each tool in Table 3.4. For both reentrancy and TOD, SAILFISH detected all the vulnerabilities (TP) with zero missed detection (FN), while maintaining the lowest false positive (FP) rate. We discuss the FPs and FNs of the tools in the subsequent sections.

False positive analysis. While reasoning about the false positives generated by different tools for the reentrancy bug, we observe that both VANDAL and OYENTE consider every

external call to be reentrant if it can be reached in a recursive call to the calling contract. However, a reentrant call is *benign* unless it operates on an inconsistent state of the contract. SECURIFY considers SOLIDITY `send` and `transfer` APIs as external calls, and raisesd violation alerts. Since the gas limit $(2,300)$ for these APIs is inadequate to mount a reentrancy attack, we refrain from modeling these APIs in our analysis. Additionally, SECURIFY failed to identify whether a function containing the external call is access-protected, *e.g.*, it contains the `msg.sender == owner` check, which prohibits anyone else but only the contract owner from entering the function. For both the cases above, though the EXPLORER detected such functions as potentially unsafe, the benefit of symbolic evaluation became evident as the REFINER eliminated these alerts in the subsequent phase. MYTHRIL detects a state variable read after an external call as malicious reentrancy. However, if that variable is not written in any other function, that deems the read *safe*. Since SAILFISH looks for *hazardous access* as a pre-requisite of reentrancy, it does not raise a warning there. However, SAILFISH incurs false positives due to imprecise static taint analysis. A real-world case study of such a false positive is presented in Appendix **??**.

To detect TOD attacks, SECURIFY checks for *writes* to a storage variable that influences an Ether-sending external call. We observed that several contracts flagged by SECURIFY have storage writes inside the contract's constructor. Hence, such writes can only happen once during contract creation. Moreover, several contracts flagged by SECURIFY have both storage variable writes, and the Ether sending external call inside methods which are guarded by predicates like `require(msg.sender == owner)`—limiting access to these methods only to the contract owner. Therefore, these methods cannot be leveraged to launch a TOD attack. SAILFISH prunes the former case during the EXPLORE phase itself. For the latter, SAILFISH leverages the REFINE phase, where it finds no difference in the satisfiability of two different symbolic evaluation traces. Next, we present a real-world case where both SECURIFY and SAILFISH incur a false positive due to insufficient reasoning of

contract semantics.

Figure 3.13 features a real-world contract where `bTken` is set inside the constructor. The static taint analysis that SAILFISH performs disregards the fact that Line 5 is guarded by a `require` clause in the line before; thereby making the variable tainted. Later at Line 9 when the `balanceOf` method is invoked on `bTken`, SAILFISH raises a false alarm.

```
1  contract EnvientaPreToken {
2    // Only owner can set bTken
3    function enableBuyBackMode(address _bTken) {
4      require( msg.sender == _creator );
5      bTken = token(_bTken);
6    }
7    function transfer(address to, uint256 val) {
8      // Trusted external call
9      require(bTken.balanceOf(address(this))>=val);
10     balances[msg.sender] -= val;
11   }
12 }
```

Figure 3.13: False positive of SAILFISH (Reentrancy).

```
1 contract Depay{
2     function pay(..., uint donation) {
3         donations += donation;
4     }
5     function withdrawDonations(address recipient) {
6         require(msg.sender == developer)
7         recipient.transfer(donations);
8     }
9 }
```

Figure 3.14: False positive of TOD.

Figure 3.14 presents a real-world donation collection contract, where the contract transfers the collected donations to its recipient of choice. Both SAILFISH and SECURIFY raised TOD warning as the transferred amount, *i.e.*, `donations` at Line 7, can be modified by function `pay()` at Line 3. Though the amount of Ether withdrawn (`donations`) is

different depending on which of `withdrawDonations()` and `pay()` get scheduled first—this does not do any harm as far as the functionality is concerned. In fact, if `pay()` front-runs `withdrawDonations()`, the recipient is rewarded with a greater amount of donation. Therefore, this specific scenario does not correspond to a TOD attack.

False negative analysis. SECURIFY missed valid reentrancy bugs because it considers only Ether sending call instructions. In reality, any call can be leveraged to trigger reentrancy by transferring control to the attacker if its destination is tainted. To consider this scenario, SAILFISH carries out a taint analysis to determine external calls with tainted destinations. Additionally, SECURIFY missed reentrancy bugs due to lack of support for destructive write (DW), and delegate-based patterns. False negatives incurred by MYTHRIL are due to its incomplete state space exploration within specified time-out. Our manual analysis did not observe any missed detection by SAILFISH.

Finding zero-day bugs using Sailfish. In order to demonstrate that SAILFISH is capable of finding zero-day vulnerabilities, we first identified the contracts flagged only by SAILFISH, but no other tool. Out of total 401 reentrancy-only and 721 TOD-only contracts, we manually selected 88 and 107 contracts, respectively. We limited our selection effort only to contracts that contain at most 500 lines of code, and are relatively easier to reason about in a reasonable time budget. Our manual analysis confirms 47 contracts are *exploitable* (not just *vulnerable*)—meaning that they can be leveraged by an attacker to accomplish a malicious goal, *e.g.*, draining Ether, or corrupting application-specific metadata, thereby driving the contract to an unintended state. We present a few vulnerable patterns, and their exploitability.

We have redacted the code, and masked the program elements for the sake of anonymity and simplicity. The fact that the origin of the smart contracts can not be traced back in most of the cases makes it hard to report these bugs to the concerned developers. Also, once a contract gets deployed, it is difficult to fix any bug due to the immutable nature of

the blockchain.

Cross-function reentrancy: Figure 3.15 presents a simplified real-world contract—vulnerable to *cross-function* reentrancy attack due to Destructive Write (DW). An attacker can set both `item_1.creator` (Line 11), and `item_1.game` (Line 12) to an arbitrary value by invoking `funcB()`. In `funcA()`, an amount `amt` is transferred to `item_1.creator` through `transferFrom`—an untrusted external contract call. Therefore, when the external call is underway, the attacker can call `funcB()` to reset both `item_1.creator`, and `item_1.game`. Hence, `item_1.fee` gets transferred to a different address when Line 6 gets executed.

```
1  function funcA(to, amt) public {
2      ...
3      IERC721 erc721 = IERC721(item_1.game)
4      erc721.transferFrom(_, item1.creator, amt)
5      ...
6      item1.creator.transfer(item_1.fee)
7  }
8
9  function funcB(_creator, _game) {
10     ...
11     item_1.creator = _creator
12     item1_1.game = _game
13     ...
14 }
```

Figure 3.15: Real-world cross-function reentrancy

Delegate-based reentrancy: Figure 3.16 presents a real-world contract, which is vulnerable to delegate-based reentrancy attack.

The contract contains three functions—(a) `funcA` contains the `delegatecall`, (b) `funcB()` allows application data to be modified if the assertion is satisfied, and (c) `funcC` contains an untrusted external call. A malicious payload can be injected in the `_data` argument of `funcA`, which, in turn, invokes `funcC()` with a tainted destination `_to`. The `receiver` at Line 14 is now attacker-controlled, which allows the attacker to reenter to `funcB` with `_isTokenFallback` inconsistently set to `true`; thus rendering the assertion at Line 8 useless.

```
function funcA(bytes _data) {
    __isTokenFallback = true;
    address(this).delegatecall(_data);
    __isTokenFallback = false;
}

function funcB(){
    assert(__isTokenFallback);
    // Write to application data
}

function funcC(address _to) {
    Receiver receiver = Receiver(_to)
    receiver.tokenFallback(...)
    ...
}
```

Figure 3.16: Real-world delegatecall-based reentrancy

CREAM Finance reentrancy attack. By exploiting a reentrancy vulnerability in the CREAM Finance, a decentralized lending protocol, the attacker stole 462,079,976 AMP tokens, and 2,804.96 Ethers on August 31, 2021 [94]. The attack involved two contracts: CREAM Finance contract `C`, and `AMP` token (ERC777) contract `A`. The `borrow()` method of `C` calls the `transfer()` method of `A`, which, in turn, calls the `tokenReceived()` hook of the receiver contract `R`. Such a hook is simply a function in `R` that is called when tokens are sent to it. The vulnerability that the attacker leveraged is that there was a state (S) update in `C.borrow()` following the external call to `A.transfer()`. Since, `A.transfer()` further calls `R.tokenReceived()` before even the original `C.borrow()` call returns, the attacker took this opportunity to reenter `C` before even the state update could take place.

Since the version of SLITHER that SAILFISH uses lacks support for all types of SOLIDITY tuples, we could not run our tool as-is on the contract `C`. To see whether our approach can still detect the above vulnerability by leveraging its inter-contract analysis, we redacted the contracts to eliminate syntactic complexity unrelated to the actual vulnerability. When run on the simplified contract, SAILFISH successfully flagged it as vulnerable to the reentrancy attack, as expected.

Transaction order dependency: TOD may enable an attacker to earn profit by

front-running a victim's transaction. For example, during our manual analysis, we encountered a contract where the contract owner can set the price of an item on demand. A user will pay a higher price for the item if the owner maliciously front-runs the user's transaction (purchase order), and sets the price to a higher value. In another contract that enables buying and selling of tokens in exchange for Ether, the token price was inversely proportional with the current token supply. Therefore, an attacker can front-run a buy transaction T, and buy n tokens having a total price p_l. After T is executed, the token price will increase due to a drop in the token supply. The attacker can then sell those n tokens at a higher price, totaling price p_h, and making a profit of $(p_h - p_l)$. We illustrate one more real-world example of a TOD attack in Figure 3.17 . `recordBet()`

```
1  contract Bet {
2   function recordBet(bool bet, uint _userAmount) {
3    userBlnces[msg.sender]= _userAmount;
4    totalBlnc[bet] = totalBlnc[bet] +_userAmount;
5   }
6   function settleBet(bool bet) {
7    uint reward = (userBlnces[msg.sender]*totalBlnc[!bet]
8                   / totalBlnc[bet];
9    uint totalWth = reward + userBlnces[msg.sender];
10   totalBlnc[!bet] = totalBlnc[!bet] - reward;
11   msg.sender.transfer(totalWth);
12  }
13 }
```

Figure 3.17: Real-world example of a TOD bug.

allows a user to place a bet, and then it adds (Line 4) the bet amount to the total balance of the contract. In `settleBet()`, a user receives a fraction of the total bet amount as the reward amount. Therefore, if two invocations of `settleBet()` having same `bet` value race against each other, the front-running one will earn higher reward as the value of `totalBlnc[!bet]`, which `reward` is calculated on, will also be higher in that case.

Exploitability of the bugs. We classified the true alerts emitted by SAILFISH into the following categories—An *exploitable* bug leading to the stealing of Ether, or application-

Tool	Small	Medium	Large	Full
SECURIFY	85.51	642.22	823.48	196.52
VANDAL	16.35	74.77	177.70	30.68
MYTHRIL	917.99	1,046.80	1,037.77	941.04
OYENTE	148.35	521.16	675.05	183.45
SAILFISH	9.80	80.78	246.89	30.79

Table 3.5: Analysis times (in seconds) on four datasets.

specific metadata corruption (*e.g.*, an index, a counter, *etc.*), and a *non-exploitable* yet vulnerable bug that can be reached, or triggered (unlike a false positive), but its side-effect is not persistent. For example, a reentrant call (the attacker) is able to write to some state variable V in an unintended way. However, along the flow of execution, V is overwritten, and its *correct* value is restored. Therefore, the effect of reentrancy did not persist. Another example would be a state variable that is incorrectly modified during the reentrant call, but the modification does not interfere with the application logic, *e.g.*, it is just written in a log. Out of the 47 zero-day bugs that SAILFISH discovered, 11 allow an attacker to drain Ethers, and for the remaining 36 contracts, the bugs, at the very least (minimum impact), allow the attacker to corrupt contract metadata—leading to detrimental effects on the underlying application. For example, during our manual analysis, we encountered a vulnerable contract implementing a housing tracker that the allowed addition, removal, and modification of housing details. If a house owner adds a new house, the contract mandates the old housing listing to become inactive, *i.e.*, at any point, there can only be one house owned by an owner that can remain in an active state. However, we could leverage the reentrancy bug in the contract in a way so that an owner can have more than one active listing. Therefore, these 36 contracts could very well be used for stealing Ethers as well, however, we did not spend time and effort to turn those into exploits as this is orthogonal to our current research goal.

Comparison against Sereum. Since SEREUM is not publicly available, we could only compare SAILFISH on the contracts in their released dataset. SEREUM [24] flagged total

16 contracts for potential reentrancy attacks, of which 6 had their sources available in the ETHERSCAN, and therefore, could be analyzed by SAILFISH. Four out of those 6 contracts were developed for old SOLIDITY versions (<0.3.x)—not supported by our framework. We ported those contracts to a supported SOLIDITY version (0.4.14) by making minor syntactic changes not related to their functionality. According to SEREUM, of those 6 contracts, only one (`TheDAO`) was a true vulnerability, while five others were its false alarms. While SAILFISH correctly detects `TheDAO` as *unsafe*, it raises a false alarm for another contract (`CCRB`) due to imprecise modeling of untrusted external call.

RQ1: SAILFISH emits the fewest warnings in the full dataset, and finds 47 zero-day vulnerabilities. On our manual analysis dataset, SAILFISH detects all the vulnerabilities with the lowest false positive rate.

3.7.3 Performance analysis

Table 3.5 reports the average analysis times for each of the small, medium, and large datasets along with the full dataset. As the data shows, the analysis time increases with the size of the dataset for all the tools. VANDAL [84] is the fastest analysis across all the four datasets with an average analysis time of 30.68 seconds with highest emitted warnings (52.27%). SECURIFY [25] is approximately 6x more expensive than VANDAL over the entire dataset. The average analysis time of MYTHRIL [27] is remarkably high (941.04 seconds), which correlates with its high number of time-out cases (66.84%). In fact, MYTHRIL's analysis time even for the small dataset is as high as 917.99 seconds. However, another symbolic execution based tool OYENTE [28] has average analysis time close to 19% to that of MYTHRIL, as it fails to analyze most of the medium to large contracts due to the unsupported SOLIDITY version. Over the entire dataset, SAILFISH takes as low as 30.79 seconds with mean analysis times of 9.80, 80.78, and 246.89 seconds

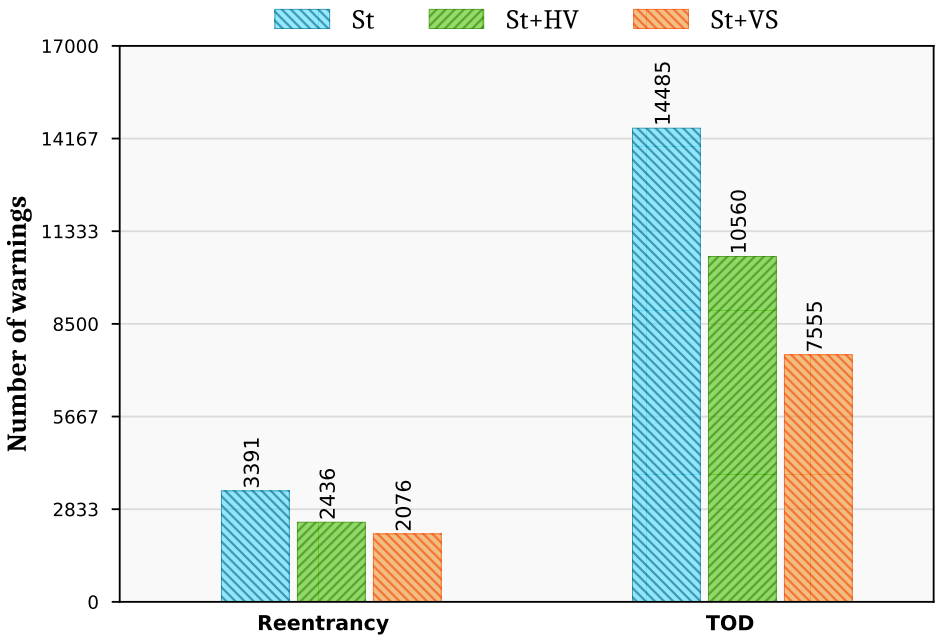

Figure 3.18: Ablation study showing the effectiveness of value-summary analysis for reentrancy and TOD detection.

for small, medium, and large ones, respectively. The mean static analysis time is 21.74 seconds as compared to the symbolic evaluation phase, which takes 39.22 seconds. The value summary computation has a mean analysis time of 0.06 seconds.

RQ2: While the analysis time of SAILFISH is comparable to that of VANDAL, it is 6, 31, and 6 times faster than SECURIFY, MYTHRIL, and OYENTE, respectively.

3.7.4 Ablation study

Benefit of value-summary analysis: To gain a better understanding of the benefits of the symbolic evaluation (REFINE) and the value-summary analysis (VSA), we performed an ablation study by configuring SAILFISH in three distinct modes: **(a)** *static-only* (**SO**), only the EXPLORER runs, and **(b)** *static + havoc* (**St+HV**), the REFINER runs, but it *havocs* all the state variables after the external call. **(c)** *static + value summary* (**St+VS**),

the REFINER runs, and it is supplied with the value summary facts that the EXPLORER computes. Figure 3.18 shows the number of warnings emitted by SAILFISH in each of the configurations. In **SO** mode, the EXPLORE phase generates 3,391 reentrancy and 14,485 TOD warnings, which accounts for 3.92% and 16.75% of the contracts, respectively. Subsequently, **St+HV** mode brings down the number of reentrancy and TOD warnings to 2,436 and 10,560, which is a 28.16% and 27.10% reduction with respect to the **SO** baseline. Lastly, by leveraging value summary, SAILFISH generates 2,076 reentrancy and 7,555 TOD warnings in **St+VS** mode, which is a 14.78% and 28.46% improvement over **St+HV** configuration. This experiment demonstrates that our symbolic evaluation and VSA are indeed effective to prune false positives.

Figure 3.19 shows a real-world contract that demonstrates the benefit of the value-summary analysis. A `modifier` in SOLIDITY is an additional piece of code which wraps the execution of a function. Where the underscore (_) is put inside the modifier decides when to execute the original function. In this example, the public function `reapFarm` is guarded by the modifier `nonReentrant`, which sets the `reentrancy_lock` (shortened as `L`) on entry, and resets it after exit. Due to the *hazardous access* (Line 14 and Line 18) detected on `workDone`, EXPLORER flags this contract as potentially vulnerable. However, the value summary analysis observes that the `require` clause at Line 7 needs to be satisfied in order to be able to modify the lock variable `L`, which is encoded as: $\texttt{L} = \{\langle \texttt{false}, \texttt{L} = \texttt{false}\rangle, \langle \texttt{true}, \texttt{L} = \texttt{false}\rangle\}$. In other words, there does not exist a program path that sets `L` to `false`, if the current value of `L` is `true`. While making the external call at Line 16, the program state is $\delta = \{\texttt{L} \mapsto \texttt{true}, ...\}$, which means that `L` is `true` at that program point. Taking both the value summary and the program state into account, the REFINER decides that the corresponding path leading to the *potential* reentrancy bug is infeasible.

```
1  interface Corn{
2    function transfer(address to, uint256 value);
3  }
4  contract FreeTaxManFarmer {
5    // Prevents re-entry to the decorated function
6    modifier nonReentrant() {
7      require(!reentrancy_lock);
8      reentrancy_lock = true;
9      _;
10     reentrancy_lock = false;
11   }
12
13   function reapFarm(address tokn) nonReentrant {
14     require(user[msg.sender][tokn].workDone > 0);
15     // Untrusted external call
16     Corn(tokn).transfer(msg.sender, ...);
17     // State update
18     user[msg.sender][tokn].workDone = 0;
19   }
20 }
```

Figure 3.19: The benefit of value-summary analysis.

RQ3: Our symbolic evaluation guided by VSA plays a key role in achieving high precision and scalability.

3.7.5 Speedup due to value-summary analysis:

To characterize the performance gain from the value-summary analysis, we have further designed this experiment where, instead of our value summary (**VS**), we provide a standard path-by-path function summary [79, 80, 81] (**PS**) to the REFINER module. From 16,835 contracts for which SAILFISH raised warnings (which are also the contracts sent to the REFINER), we randomly picked a subset of 2,000 contracts **(i)** which belong to either medium, or large dataset, and **(ii) VS** configuration finished successfully without timing out—for this experiment. We define *speedup* factor $s = \frac{t_{ps}}{t_{vs}}$, where t_m is the amount of time spent in the symbolic evaluation phase in mode m. In **PS** mode, SAILFISH timed out for 21.50% of the contracts owing to the increased cost of the REFINE phase. Figure 3.20

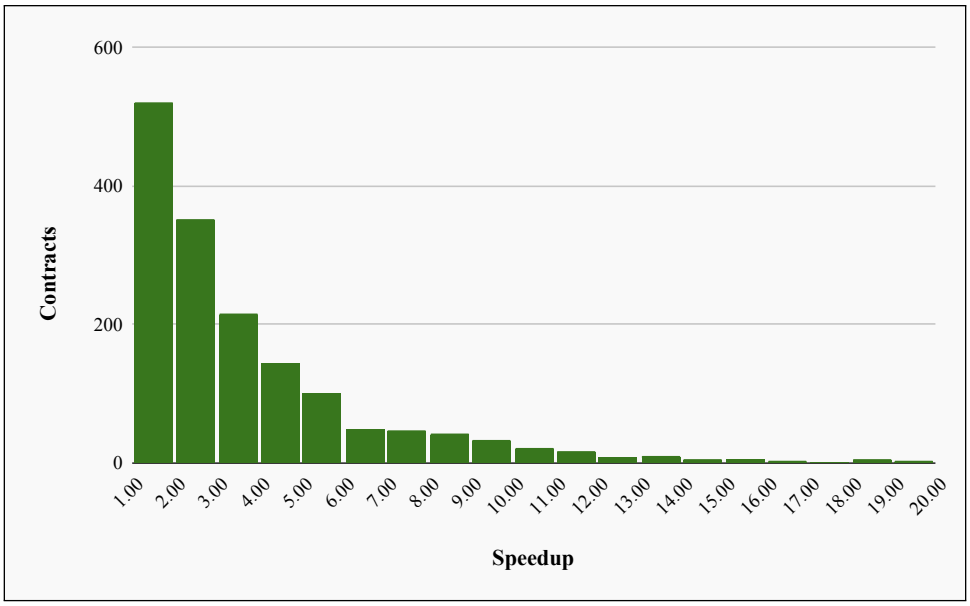

Figure 3.20: Relative speedup due to value summary over a path-by-path function summary.

presents a histogram of the speedup factor distribution of the remaining 1,570 contracts for which the analyses terminated in both the modes.

Our novel value summary analysis is significantly faster than a classic summary-based analysis.

3.8 Limitations

Source-code dependency. Although SAILFISH is built on top of the SLITHER [89] framework, which requires access to the source code, we do not rely on any rich semantic information from the contract source to aid our analysis. In fact, our choice of source code was motivated by our intention to build SAILFISH as a tool for developers, while enabling easier debugging and introspection as a side-effect. Our techniques are not tied to source code, and could be applied directly to bytecode by porting the analysis on top of a contract decompiler that supports variable and CFG recovery.

Potential unsoundness. We do not claim soundness with respect to the detection rules of reentrancy and TOD bugs. Also, the meta-language our value-summary analysis is based on distills the core features of the SOLIDITY language, it is not expressive enough to model all the complex aspects [95], *e.g.*, exception propagation, transaction reversion, out-of-gas, *etc.* In turn, this becomes the source of unsoundness of the REFINER. Additionally, SAILFISH relies on SLITHER [89] for static analysis. Therefore, any soundness issues in SLITHER, *e.g.*, incomplete call graph construction due to indirect or unresolved external calls, inline assembly, *etc.*, will be propagated to SAILFISH.

Imprecise analysis components. SAILFISH performs inter-contract analysis (Appendix **??**) when the source code of the called contract is present in our database, and more importantly, the external call destination d is statically known. If either of the conditions does not hold, SAILFISH treats such an external call as *untrusted*, thereby losing precision. The question of external call destination d resolution comes only when SAILFISH is used on contracts that have been deployed already. For cases where d is set at run-time, our prototype relies on only contract creation transactions. If d is set through a public setter method, our current prototype cannot detect those cases, though it would not be hard to extend the implementation to support this case as well. Moreover, SAILFISH incurs false positives due to the imprecise taint analysis engine from SLITHER. Therefore, using an improved taint analysis will benefit SAILFISH's precision.

Bytecode-based analysis. SAILFISH relies on control-flow recovery, taint analysis, and symbolic evaluation as its fundamental building blocks. Recovering source-level rich data structures, *e.g.*, array, strings, mappings, *etc.*, is not a requirement for our analysis. Even for EVM bytecode, recovering the entry points of public methods is relatively easier due to the "jump-table" like structure that the SOLIDITY compiler inserts at the beginning of the compiled bytecode. Typically, it is expected for a decompiler platform to provide the building blocks in the form of an API, which then could be used to port SAILFISH for

bytecode analysis. That said, the performance and precision of our analysis are limited by the efficacy of the underlying decompiler. Thanks to the recent research [96, 97, 98, 99] on EVM decompilers and static analysis, significant progress has been made in this front.

Other bugs induced by hazardous access. If a contract contains hazardous access, but no reentrancy/TOD vulnerability, that can still lead to a class of bugs called Event Ordering (EO) bugs [86], due to the asynchronous callbacks initiated from an off-chain service like `Oraclize`. We consider such bugs as out of scope for this work.

3.9 Related work

Static analysis. Static analysis tools such as SECURIFY [25], MADMAX [26], ZEUS [93], SMARTCHECK [91], and SLITHER [89] detect specific vulnerabilities in smart contracts. Due to their reliance on bug patterns, they over-approximate program states, which can cause false positives and missed detection of bugs. To mitigate this issue, we identified two complementary causes of SI bugs—Stale read and Destructive write. While the former is more precise than the patterns found in the previous work, the latter, which is not explored in the literature, plays a role in the missed detection of bugs (Section 3.2). Unlike SAILFISH, which focuses on SI bugs, MADMAX [26] uses a logic-based paradigm to target gas-focused vulnerabilities. SECURIFY [25] first computes control and data-flow facts, and then checks for compliance and violation signatures. SLITHER [89] uses data-flow analysis to detect bug patterns scoped within a single function. The bugs identified by these tools are either *local* in nature, or they refrain from doing any path-sensitive reasoning—leading to spurious alarms. To alleviate this issue, SAILFISH introduces the REFINE phase that prunes significant numbers of false alarms.

Symbolic execution. MYTHRIL [27], OYENTE [28], ETHBMC [100], SMARTSCOPY [101], and MANTICORE [102] rely on symbolic execution to explore the state-space of the contract.

ETHBMC [100], a bounded model checker, models EVM transactions as state transitions. TEETHER [103] generates constraints along a critical path having attacker-controlled instructions. These tools suffer from the limitation of traditional symbolic execution, *e.g.*, path explosion, and do not scale well. However, SAILFISH uses the symbolic execution *only* for validation, *i.e.*, it resorts to under-constrained symbolic execution aided by VSA that over-approximates the preconditions required to update the state variables across all executions.

Dynamic analysis. While SEREUM [24] and SODA [104] perform run-time checks within the context of a modified EVM, TXSPECTOR [23] performs a post-mortem analysis of transactions. ECFCHECKER [29] detects if the execution of a smart contract is *effectively callback-free* (ECF), *i.e.*, it checks if two execution traces, with and without callbacks, are equivalent—a property that holds for a contract not vulnerable to reentrancy attacks. SAILFISH generalizes ECF with the notion of hazardous access for SI attacks. Thus, SAILFISH is not restricted to reentrancy, instead, can express all properties that are caused by state inconsistencies. Dynamic analysis tools [105, 106, 107, 108, 109] rely on manually-written test oracles to detect violations in response to inputs generated according to blackbox or greybox strategies. Though precise, these tools lack coverage—which is not an issue for static analysis tools, such as SAILFISH.

Hybrid analysis. Composition of static analysis and symbolic execution has been applied to find bugs in programs other than smart contracts. For example, SYS [110] uses static analysis to find potential buggy paths in large codebases, followed by an under-constrained symbolic execution to verify the feasibility of those paths. WOODPECKER [111] uses rule-directed symbolic execution to explore only relevant paths in a program. To find double fetch bugs in OS kernels, DEADLINE [112] employs static analysis to prune paths, and later performs symbolic execution only on those paths containing multiple reads. Several other tools [113, 114, 115, 116, 117] employ similar hybrid techniques for

testing, verification, and bug finding. Such hybrid analyses have been proved effective to either prune uninteresting paths, or selectively explore interesting parts of the program. In SAILFISH, we use static analysis to filter out interesting contracts, find potentially vulnerable paths, and compute value-summary to be used in conjunction with the symbolic execution—to achieve both scalability, and precision.

State inconsistency (SI) notions. SERIF [118] detects reentrancy attacks using a notion of trusted-untrusted computation that happens when a low-integrity code, invoked by a high-integrity code, calls back into the high-integrity code before returning. Code components are explicitly annotated with information flow (trust) labels, which further requires a semantic understanding of the contract. Then, they design a type system that uses those trust labels to enforce secure information flow through the use of a combination of static and dynamic locks. However, this notion is unable to capture TOD vulnerabilities, another important class of SI bugs. In SAILFISH, we take a different approach where we define SI bugs in terms of the side-effect, *i.e.*, creation of an inconsistent state, of a successful attack. Further, we model the possibility of an inconsistent state resulting from such an attack through hazardous access. Perez *et. al.* [119], VANDAL [84], OYENTE [28] consider reentrancy to be the possibility of being able to re-enter the calling function. Not only do these tools consider only single-function reentrancy, but also the notion encompasses legitimate (benign) reentrancy scenarios [24], *e.g.*, ones that arise due to withdrawal pattern in SOLIDITY. In addition, SAILFISH requires the existence of hazardous access, which enables us to account for cross-function reentrancy bugs, as well as model only malicious reentrancy scenarios. To detect reentrancy, SECURIFY [25] looks for the violation of the "no write after external call" (NW) pattern, which is similar to the "Stale Read" (SR) notion of SAILFISH. Not all the tools that support reentrancy bugs have support for TOD. While SAILFISH shares its notion of TOD with SECURIFY, OYENTE marks a contract vulnerable to TOD if two traces have different Ether flows.

Unlike SAILFISH for which hazardous access is a pre-requisite, OYENTE raises alarm for independent Ether flows not even related to SI.

3.10 Conclusion

In this chapter, we present SAILFISH, a scalable system designed to automatically detect state-inconsistency bugs in smart contracts. To address the challenge of analyzing complex smart contracts efficiently, we propose a hybrid approach that combines a light-weight exploration phase with a precise refinement phase based on symbolic evaluation guided by a novel value-summary analysis. The experiments conducted on Ethereum smart contracts demonstrate the effectiveness of SAILFISH in detecting two types of state-inconsistency flaws: reentrancy and transaction order dependence (TOD). The hybrid approach employed by SAILFISH, along with the value-summary analysis, contributes to its efficiency and accuracy in identifying state-inconsistency bugs. Overall, SAILFISH presents a valuable contribution to the field of smart contract analysis, offering a scalable and effective solution for detecting state-inconsistency bugs and improving the security of blockchain-based applications.

Chapter 4

Exploiting the Unfair Advantage: Investigating Opportunistic Trading in the NFT Market

Decentralized Finance (DeFi), the financial system built around blockchain, eliminates intermediaries, and provides a trustless environment to its beneficiaries. The transparency of a blockchain grants its participants access to both code (smart contracts) and data (transactions). The promise of *fairness* is built into the blockchain, by design. The trustless and transparent nature of a blockchain should not only reduce the risk of fraud and manipulation, but also create a financial environment accessible equally to all the involved parties.

Unlike traditional financial markets, DeFi largely lacks comprehensive regulatory frameworks, partly because it is challenging for regulatory authorities to establish consistent guidelines, and enforce them across jurisdictions. The lack of regulation combined with the public availability of information have enabled sophisticated "players" with advanced technical expertise to spot high-value trade opportunities, and grab them in

no time; all before the rest of the market can react. For example, in a sandwich attack, the attacker first observes a large buy/sell order placed by a victim (transparency of the order book) that could cause significant price movement for an asset. Then, the attacker places their own buy/sell orders on both sides of the original order. In effect, they not only make their trade execute at a more favorable price, but also make the victim trade execute at a worse than expected price, effectively making profit out of their loss. Since the steps involved are time-sensitive and compute-intensive, such attacks can only be run by *bots* (automated programs). Evidently, such opportunities are not available to regular users without advanced knowledge.

Existing research on cryptoeconomics focusing on fairness of the cryptocurrency market, economic risks, and abuses primarily extends in two directions—**(i)** high-frequency trading of ERC-20 tokens, *e.g.*, arbitrage [120, 35, 121], miner/block extractable value (MEV/BEV) [33, 122, 123], frontrunning [31, 30, 124], flashloan [34], sandwich [32] attacks, *etc.*, and **(ii)** manipulation of both ERC-20 and ERC-721 token markets, *e.g.*, NFT rug pull [125, 126, 127], ponzi schemes [128, 129, 130, 131], cryptocurrency pump and dump [132, 133, 134], NFT trading malpractices [135, 136, 137, 138, 139, 140, 141], *etc.* However, none of the previous research studied the landscape of opportunistic trading in the NFT (ERC-721) market. Our work fills that void.

NFTs are high value assets. However, ERC-20 tokens predate NFTs in terms of inception and popularity, which is why the opportunistic trading in the NFT market has not received enough attention yet, even though the instances of such trades are equally prevalent in both markets. For example, somebody made $100K through speculative trading of CryptoKitties, and roughly $8K by running an arbitrage bot [142]. On a separate occasion, a sniper bot launched a flashloan attack by front-running a CryptoPunks bid transaction, and bagged a punk almost at no cost [143] as the reward.

Despite exhibiting opportunities similar to the traditional ERC-20 market, NFTs

come with their unique challenges for both the trader and the analyst—**(i) Determining profitability.** NFTs being non-fungible by design, no two NFTs are the same, unlike fungible ERC-20 tokens. This makes it hard to determine the "true" price of a token, which is heavily influenced by the market sentiment. Knowing the worth of an NFT is crucial for a buyer to make decisions, *e.g.*, if it is worth to buy an NFT, if it is better to sell it immediately (probably to ride on the hype), or hold it for a longer time expecting a price appreciation, *etc.* **(ii) Diverse trade actions.** Unlike ERC-20 tokens that support limited actions like buy/sell, NFT trade actions are diverse, for example, listing, auctions (place/accept/cancel bids), *etc.* Since these actions are oftentimes marketplace-specific, it is hard to build an infrastructure (*e.g.*, bot) that supports all the existing and newer marketplaces. **(iii) Spotting trading opportunity.** Typically, a ERC-20 trade opportunity spans across multiple similar type of protocols. For example, an arbitrage can be spotted by detecting cycles in token exchange rates across decentralized exchanges [120]. Where as, as we will see, profit-making trades involving NFTs typically touch multiple different types of protocols. Detecting such opportunities in real-time is hard in a competitive market. As a side effect, a "supposedly" open (and fair) market turns out to be profitable only to advanced traders.

In this work, we study the high-frequency, opportunistic trades in the NFT market. First, we taxonomize the opportunistic trades. Next, we build models that identify previous instances of such trades, and quantify their prevalence, and financial impact. By shedding light on the underexplored or unexplored aspects of the money-making opportunities, we hope that our work aims will pave the way for future research and exploration in this field.

4.1 Background

In this section, we will discuss concepts pivotal to understanding the rest of the work.

Blockchain and smart contracts. As a distributed digital ledger, the blockchain provides the necessary underpinning for cryptocurrency, a form of ownable digital asset. Transactions change the state of the blockchain. Any connected node in the network can broadcast transactions. Since the participating nodes are geographically distributed, they receive those transactions at different times and in different orders. Therefore, a consensus protocol is necessary to achieve an agreement about the ledger's state. Cryptocurrency trading relies on blockchain technology for recording transactions. The native currency of a blockchain, *e.g.*, Ether in case of the Ethereum blockchain, is intertwined with the functioning of the blockchain itself, such as sending transactions, offering incentives, and governance. In addition to storing records, some blockchains can run Turing-complete programs called smart contracts.

Ethereum blockchain. Ethereum is one of the most popular blockchains in the world. The current version (v2.0) of Ethereumfollows a consensus mechanism called *proof of stake* (PoS). An *account* is an entity represented by an address that can send transactions. An externally owned account (EOA) is controlled by anyone holding the corresponding private key, while contract accounts are managed by smart contracts. In PoS, transactions are validated by special nodes called *validators*, who stake (lock in) Ethers in the network to participate in the validation process. To incentivize the validators and prevent denial-of-service (DoS) attacks, transactions cost *gas* which is deducted from the sender's Ether balance.

Ethereumsmart contracts are written in languages like Solidity, and run on top of EthereumVirtual Machine (EVM). When a contract executes, it can emit *events* to indicate the occurrence of certain action. As shown in Figure 4.1, when `amount` worth of

funds are sent to an address `to`, a contract can emit a `fundSent` event. Events are stored in transaction logs. An event parameter can be marked as *indexed* to allow for efficient searching and filtering. In our example, `to` is indexed, while *amount* is not. Events in the transaction log of a contract are associated with a monotonically increasing identifier called *log index*, which is assigned in the order the events are emitted.

```
event fundSent(address indexed to, uint amount);
```

Figure 4.1: An event declaration in SOLIDITY.

Decentralized finance (DeFi). The smart contracts play two important roles to further the economic framework built around blockchain—**(a)** they create and manage digital assets, called *tokens*. For example, Tether (USDT) is a fungible ERC-20 token, while DECENTRALAND is a non-fungible ERC-721 token. In the same way as the primary (native) currency, these secondary assets are also equally ownable and tradable. **(b)** they implement decentralized protocols to automate financial processes, *e.g.*, lending and borrowing protocols, decentralized exchanges (DeX), *etc.* This open financial system built around blockchain is called Decentralized Finance (DeFi).

Non-fungible token (NFT). All copies of a fungible token are identical, while every non-fungible token (NFT) is unique. Among the NFTs managed by a smart contract, each NFT is uniquely identified by a `tokenId`. The creation and destruction of a token is referred to as *minting* and *burning*, respectively. ERC-721 and ERC-1155 are two popular standards for NFTs. While ERC-721 allows creation of only unique tokens, ERC-1155 is a multi-token standard that supports both fungible and non-fungible tokens within the same contract. In effect, an NFT (identified by a `tokenId`) can have only one copy in a ERC-721 contract, while it can have multiple copies in ERC-1155. The `totalSupply` variable in a contract keeps track of the number of unique NFTs in circulation. When a token with `tokenId` is transferred from the `from` address to the `to` address, a `Transfer`

event (Figure 4.2) is emitted on the log. A *collection* refers to a set of NFTs that are managed by one smart contract, and typically have something in common, for example, they may share certain attributes, or be used for the same purpose. NFTs in one collection typically look alike, such as CryptoPunks.

```
event Transfer(address indexed from,
               address indexed to, uint256 tokenId);
```

Figure 4.2: `Transfer` event in ERC-721 and ERC-1155.

NFT marketplace. NFT marketplaces (NFTMs) like OpenSea, SuperRare, Sorare connect sellers to buyers. The sellers either list (T1) their items (NFTs) for sale, or put them on auction (T2). The buyers, in turn, can either buy (T3) the items at the listed price, or make offers (T4), or place bids (T5) if the item is on auction. Should they change their mind, the buyers (bidders) can also retract (T6) their bids if they have not been filled yet. Sometimes, instead of making an offer on an individual NFT, buyers can make a *collection offer* (T7) for all NFTs in a collection. Collection offers are useful if a buyer would like to buy any NFT in a collection, but does not have a specific NFT in mind. A given collection offer can only be accepted once, and will expire for all other NFTs in the collection after being accepted. We call T1-7 *trade actions*.

NFT liquidity pool. The NFTMs discussed above follow a traditional orderbook-based model, where the platform maintains the list of all open orders. Instead of directly connecting buyers to sellers, automated market markers (AMMs), also known as decentralized exchanges (DEX), facilitate near-instant token swaps by algorithmically setting prices for assets based on supply and demand. Funds are sold to and bought from liquidity pools (LP), which are smart contracts that lock up large volume of crowdsourced tokens. Examples of popular ERC-20 DEXes are Uniswap, Curve, Balancer, *etc.* Similarly, an NFT AMM is a platform that allows traders to instantly buy or sell their

NFTs through LPs. While traditional NFTMs allow the seller to have better control on the price of the item to be sold, NFT LPs are better suited for cases where the seller wants to make an instant trade.

Let's take NFTX [144] as an example, which is a constant-product AMM. On NFTX, NFT holders deposit their NFTs into a vault, and mint a fungible token (vToken) specific to that NFT collection, for example, PUNK for CryptoPunks, in return. That "vToken" that represents a 1:1 claim on a random NFT from within that collection's vault. If and when the user wants to claim any NFT from the vault, they can bring the token back, and redeem an NFT from the collection. Note that the vToken represents a claim on a random NFT from the specific collection, not the exact one the holder deposited.

LPs consist of a token pair, where one token can be exchanged for the other. vTokens can be deposited to an AMM like Sushiswap to create a liquid market. Consider such a vToken-ETH pool with 10 vTokens and 2 ETH. This would make the price of the corresponding NFT $2/10 = 0.20$ ETH. Now, if one NFT is purchased for 0.20 ETH, that NFT is removed from one side of the pool, while 0.20 ETH get added onto the other side, resulting in 9 NFTs and 2.2 ETH. With this, the price of the NFT goes up to $2.2/9 = 0.24$ ETH.

4.2 Analysis approach

This paper studies opportunistic trading in the NFT market. Traditional trading strategies tend to be based on fundamental analysis, such as studying a company's financial statements or economic indicators, and making investment decisions based on the expected long-term value of an asset. On the other hand, opportunistic trading relies on trends and patterns in the market. It differs from traditional trading strategies in that

it mostly seeks to take advantage of short-term market inefficiencies, news events, hype, volatility, temporary imbalances in supply and demand, price discrepancies, *etc.*, rather than attempting to profit from long-term trends. While traditional trading is "proactive," where the trader strives to anticipate long-term market conditions, opportunistic trading is far more "reactive," where a trader is forced to react quickly to instantaneous market situations.

In this paper, we attempt to answer the following research questions—**(RQ1.)** What types of opportunistic trading instances are prevalent in the NFT market? **(RQ2.)** What is the financial impact of each type of opportunistic trade? **(RQ3.)** How does opportunistic trading differ in the NFT space from the more traditional ERC-20 and cryptocurrency market?

Attacker model. An opportunistic trader performs the following two functions—**(i) Spotting opportunities.** Like many other blockchain-based financial protocols, NFT protocols are complex as well. This is even more true with the rise of newer financial instruments, such as NFT liquidity pools. To exacerbate the situation, opportunities often arise from the composition of more than protocols. For example, an arbitrageur takes advantage of the discrepancies of the price of an asset (NFT) across different exchanges. Since such opportunities are short-lived, they often need to be spotted in real-time. **(ii) Executing trades.** Opportunities are competitive, thereby needing the trader to quickly analyze, and react to changing market conditions in order to capitalize on short-term opportunities.

In our model, an attacker is an opportunistic trader with solid financial and technical knowledge, and access to reasonable computing resources to perform both the above functions. Therefore, most of the instances as we will see next are high-frequency, bot-driven trades. Depending on the type and timing of the trade, the attacker can be both a seller and a buyer.

Methodology and data collection. We perform both qualitative and quantitative analysis to understand the landscape. Since manually analyzing the profitability and the rationale behind all NFT trades is not feasible, it makes the problem of finding opportunistic trading challenging. To address this issue to the best possible extent, we gathered instances of opportunistic trading both actively (finding ourselves) and passively (finding previous reports). For the passive analysis, we collected previously reported instances from publicly available resources, such as, blogs, forums, write-ups and TWITTER keyword search. Specifically, we searched TWITTER with related hashtags, *e.g.*, front-run, mev, arbitrage, *etc.*, and then manually filtered out irrelevant results. Additionally, to discover new instances, we randomly sampled 100 trades from the NFT sales over $5,000 USD for each of the NFTMs we support, and manually analyzed them for the sign of suspicious, opportunistic trades.

We identify three broad categories of trades—**(i) Acquire.** These strategies are used to grab high-value NFTs that are worth holding for a long time (Section 4.3). **(ii) Profit generation.** These strategies are used to generate instant profit through simultaneous buying and selling of NFTs (Section 4.4). **(iii) Loss minimization.** These strategies are used to minimize losses from a potentially bad purchase (Section 4.5).

▶ **Marketplace selection.** To include an NFTM in our analysis, we had to first understand the marketplace-specific protocol, and then develop protocol-specific transaction parsers, which is non-trivial. Therefore, we carefully integrated the support for the most prominent NFTMs in our analysis. In line with the previous work [145, 135], we used DAPPRADAR [146], a popular tracker for dApps, to select the relevant marketplaces. Due to the popularity of Ethereum, we selected the top four (as on July 15, 2022) Ethereum-backed NFTMs ranked by all-time transaction volume—OPENSEA (34.69B), LOOKSRARE (4.84B), CRYPTOPUNKS (3.1B), and X2Y2 (1.24B). Further, we support all three versions of OPENSEA marketplace, *viz.*, v1.0, v2.0, and v3.0. In addition, during our passive

analysis, we discovered numerous reports of opportunistic trading in CryptoKitties (30.1M), which is one of the older and popular collection. We included that too in our analysis.

▶ **Recovering trade actions.** For the NFTMs we considered in this work, relevant NFT trade actions (Section 4.1) are visible from the blockchain. The quantitative analysis was performed on the Ethereum blockchain data until March 15, 2023 (block 16,000,000) from the genesis block.

We process each block B starting from the genesis block ($block\ number = 0$). Now, such opportunistic trades can be executed either through an EOA, or if the trade involves complicated run-time logic, touches multiple protocols, and requires atomicity of a batch of transactions, then through a *bot* contract. Therefore, in order to conduct a comprehensive analysis, we consider both external and internal transactions, which we denote as T.

$\mathsf{t\_info}(b, h, st, s, r,$ $gp, gu, ind, d, ts)$	$:-$	Transaction at block b with hash h, status st, sender s, receiver r, gas price gp, timestamp ts index ind and gas used gu
$\mathsf{listing}(u, m, n, p)$	$:-$	User u lists an NFT n at NFTM m at price p
$\mathsf{cancel\_listing}(u, m, p, n)$	$:-$	User u cancels the listing of an NFT n listed at NFTM m at price p
$\mathsf{buy}(u, m, p, n)$	$:-$	User u buys an NFT n listed at NFTM m at price p
$\mathsf{place\_bid}(u, m, p, n)$	$:-$	User u placed a bid of value p on an NFT n listed at NFTM m
$\mathsf{accept\_bid}(u, m, p, n)$	$:-$	User u accepts a bid of value p on an NFT n listed at NFTM m
$\mathsf{cancel\_bid}(u, m, p, n)$	$:-$	User u cancels a bid of value p on an NFT n listed at NFTM m

Figure 4.3: Extracted predicates from transactions

NFTMs vary in their trading functionalities, and may even have distinct implementations for the same trade actions. Therefore, we need to consider marketplace-specific

implementation to infer NFT trade details, such as action type, targeted tokens, their prices, *etc.*, during our analysis. Figure 4.3 illustrates the predicates we extract from a transaction T. For each $T \in B$, we extract two types of information—**(i)** generic transaction details, such as block number b, transaction hash h, transaction status st, sender s and receiver r, gas price gp, gas used gu, transaction index ind within the containing block B, transaction data d, timestamp ts, *etc.*, and **(ii)** NFT trade-specific information, such as trade actions, details n of the NFT involved in this trade, marketplace m where the trade takes place, the buy/sell/bid price p of n, *etc.*

We extract generic transaction details from the blockchain, which is then used to infer trade-specific information in the following two steps— **(Step-I)** We first understand the NFTM protocol by examining the source code of the marketplace contract, and observing past transactions. Thus, we identify the public methods corresponding to the trade actions, and understand the semantics of their arguments. **(Step-II)** In SOLIDITY, each method (having a specific signature) is hashed to a unique four-byte-long string called *method selector*. The selector is encoded in the transaction data when a method is invoked. Therefore, we match the selectors that appear in the decoded transaction data d with the ones identified in the previous step to infer trade actions.

In addition, for a transaction T we define the following operator—`TradeAction(T)`: returns the trade action if T represents one of the supported actions, *viz.*,
$\mathcal{A} = \{\mathsf{listing}, \mathsf{cancel\_listing}, \mathsf{buy}, \mathsf{place\_bid}, \mathsf{accept\_bid}, \mathsf{cancel\_bid}\}$, *null* otherwise. For a particular type of trade transaction T, we use the *dot* (.) notation to refer to its attributes as defined in Figure 4.3. For example, if T_l is a listing transaction, $T_l.m$ denotes the marketplace where the respective NFT is listed at, and so on. Finally, $\mathcal{T}$ denotes the set of transactions of our interest across all the supported NFTMs $\mathcal{M}$, *i.e.*, $\mathcal{T} = \{T \mid T.m \in \mathcal{M} \wedge \texttt{TradeType(T)} \in \mathcal{A}\}$. We use the notations, predicates, and operators introduced above throughout the rest of this paper.

▶ **Token contract identification.** We built a dataset of ERC-721 and ERC-1155 token contracts (contract addresses) for our analyses. Our method of token contract identification does not require the source code of the contracts. Since only a fraction of the contracts on the blockchain have their sources available [147], a source code-agnostic method significantly broadens the scope of our analysis. We took a two-step approach— **(Step-I)** We relied on the fact that a ERC-721- and ERC-1155-conforming token always emits a `Transfer` event (Section 4.1) when an NFT is transferred from one account to the other. Such `Transfer` event logs can be identified from the blockchain in a way similar to the method selector-based approach (used for method identification) described above. **(Step-II)** Unfortunately, ERC-20 token standard, too, defines a similar `Transfer` event upon the transfer of token, thus making their event logs "apparently" indistinguishable from the ones emitted by the NFT contracts. In effect, the `Transfer` logs identified in the previous step contain false positives, which includes the ones emitted by ERC-20 tokens as well. Now, for ERC-20 token, the third (last) argument of the `Transfer` event represents the `value` (amount) of the tokens transferred, while for both the NFT standards, it represents the `tokenId`. As defined by the respective standards, this particular argument is *indexed* only for the NFTs, but not for the ERC-20 tokens. We leverage this observation to identify the `Transfer` logs emitted by the NFT contracts, and record the corresponding addresses.

In the end, we identified 152,478 ERC-721 and 27,824 ERC-1155 token contracts in total. Further, we randomly sampled 200 of the identified contracts, and verified if those are, indeed, NFTs by consulting both OpenSea and Etherscan. We found that 195 (97.50%) of them are true positives.

▶ **Price feed.** We fetch historical price data from CoinGecko [148] to convert token (cryptocurrency) prices to equivalent US Dollars at the time of the respective trades.

4.3 Acquire

Frontrunning refers to the malicious practice of exploiting the knowledge of a pending victim transaction T_v, and use that knowledge to execute an attack transaction T_a before T_v. By scheduling their transaction before victim's, an attacker makes a profit that the victim was supposed to make, leaving the unfortunate victim suffer a loss.

The Ethereummempool is a globally visible, temporary, public storage area of unconfirmed transactions that are waiting to be included in a block. When a user sends a transaction, it enters the mempool, and awaits validation by miners. Miners typically prioritize transactions according to the gas price offered. The higher the gas price, the more incentive miners have to include the transaction in the next block they mine. In frontrunning, an attacker first observes a victim transaction T_v from the mempool. If it deems profitable, then they replay the same transaction, but with a higher gas price, so that the miners include the attack transaction T_a before T_v. Ironically, in context of opportunistic trading, the attacker not only banks on the victim's effort of finding a profitable trade opportunity (which is already hard in the competitive market), but also robs the victim of their profit.

With the rising popularity of NFTs as an investment vehicle, frontrunning has become prevalent in the NFT trading as well. Traders frontrun NFT trades to either buy or sell NFTs depending on the trade actions (Section 4.1) performed by the trades. We identified the following types of frontrunning attacks in the NFT trading.

Buy–Buy. In this attack, both the victim V and the attacker A are buyers. When A spots a profitable buy order transaction T_v submitted by V to purchase an NFT N, A frontruns T_v with another buy order T_a targeting the same NFT. As a result, A acquires the NFT, while T_v becomes invalid and eventually reverts.

Interestingly, in theory, if A knew beforehand that purchasing N is a worthy investment,

then they could probably make the same purchase without even frontrunning V, just like a regular buyer. However, NFT market is hype-driven, and speculative. Only a few special-purpose utility NFTs which represent in-game assets, or tickets to access services, hold intrinsic value. Except those ones, NFT valuation is largely derived from the external factors like supply and demand, or people's perception. Seeing the victim V attempting to purchase an NFT serves as an "endorsement" of public interest, which is probably what motivates A to frontrun the victim.

Buy–Cancel. In this attack, the victim V is a seller, while the attacker A is a buyer. When A observes a cancel order transaction T_v to cancel the listing of an NFT submitted by V, A frontruns T_v with a buy order T_a targeting the same NFT. As a result, A acquires the NFT, while T_v becomes invalid and eventually reverts.

Typically, this type of frontrunning attack happens when a seller unintentionally lists an NFT at an exceptionally lucrative price, but then realizes the mistake, and promptly attempts to cancel the listing to avoid any potential loss. Unfortunately, this very attempt of the seller to prevent a loss attracts the attention of the attacker to that NFT, which would otherwise probably remain unnoticed.

Accept bid–Cancel bid. In this attack, the victim V is a buyer, while the attacker A is a seller. When A observes a cancel bid transaction T_v to cancel a prior bid submitted by V on an NFT, A frontruns T_v with an accept bid transaction T_a, thus forcing V to purchase the NFT against their will.

This can occur either due to the buyer's accidental bid placement, or their subsequent realization that the deal may not be beneficial. The seller then act quickly to exploit the situation before the buyer's bid is canceled, ensuring they do not miss out on the possibility of a profitable outcome.

Place bid–Accept bid. In this attack, both the victim V and the attacker A are buyers. When A observes an accept bid transaction T_{ab} from a seller to accept a bid B previously

submitted by a potential buyer V, A frontruns T_{ab} with a place bid transaction T_a with a slightly higher bid amount than B, so that they the NFT is sold to them instead of the victim.

In certain cases involving popular collections, there have been instances where victim buyers place extremely low bid B, most likely to try out their luck. Then they patiently wait for the sellers to accept those bids. On the other hand, the seller gets frustrated by only receiving low bids, and eventually goes ahead to accept one of these low bids. An attacker then exploits the situation by frontrunning the accept bid transaction T_{ab} by placing a slightly higher bid than the existing one. As a consequence, the original bid is removed from consideration, and the attacker's bid becomes the most recent one. Consequently, the seller ends up accepting the bid placed by the attacker instead of the initial bid B placed by the victim buyer. Note that this attack is only possible if the marketplace protocol does not include any unique identifier corresponding to the victim bid B in T_{ab}. Rather, T_{ab} bid-agnostically accepts the highest bid at any point, which is exploited by the attacker. Unlike other cases where the victim transaction fails as the side-effect of the frontrun, here it is necessary that both transactions execute successfully for the attack to work.

Identifying frontrunning attacks. To quantify the prevalence of the frontrunning attacks presented above, we apply the following set of rules to detect past instances of successful frontruns.

Let $T_1, T_2 \in \mathcal{T}$ be two transactions, such that $T_1.ts \leq T_2.ts$. Then, we say that T_1 frontruns T_2, if:

Rule 1. They involve the same NFT, and the same marketplace, but come from different senders. Formally, $T_1.n = T_2.n$, and $T_1.m = T_2.m$, and $T_1.s \neq T_2.s$.

Rule 2. The transactions are "competing" with each other. Either T_1 appears before T_2 in the same block, *i.e.*, $T_1.b = T_2.b$ and $T_1.ind < T_2.ind$, or T_1 appears in the block

Strategy	Marketplaces				
	OpenSea	LooksRare	CryptoPunks	X2Y2	CryptoKitties
Acquire					
Frontrunning					
Buy–Buy	274,129	1,459	53	40,623	3,549
Buy–Cancel	4,640	0	2	7	33
Accept bid–Cancel bid	167	654	3	0	0
Place bid–Accept bid	0	0	200	0	0
Backrunning					
Listing–Buy	0	0	130	0	181
Loss minimization					
Cancel–Buy	1,396	83	8	356	25

Table 4.1: Frontrunning and backrunning instances found in different marketplaces

immediately before T_2, *i.e.*, $T_2.b = T_1.b + 1$. In both cases, gas price offered for T_1 is higher than that of T_2, *i.e.*, $T_{1_{gp}} > T_{2_{gp}}$.

Rule 3. The transactions represent appropriate type of trade required by the type of frontrun in question. For example, if we are looking for a Buy–Cancel frontrun, then $\texttt{TradeType}(T_1) = \mathsf{cancel\_listing}$ and $\texttt{TradeType}(T_2) = \mathsf{buy}$.

Rule 4. As discussed earlier, for all types of frontrun categories except the last one (Place bid–Accept bid), the victim transaction fails, while the attack transaction succeeds. Therefore, we perform appropriate checks on $T_1.st$ and $T_2.st$ status values.

Limitations. Our mechanism detects *competing* transactions by comparing the gas prices of the victim and attack transactions (Rule 2). In PoS Ethereum, MEV Boost [149] offers a frontrunning-protection service through private mining. In effect, the users can now make direct on-chain payments to the validator to incentivize the inclusion of a transactionin a block, instead of participating in the usual gas-auction. Attackers can use MEV Boost or similar protocols not only to bypass the mempool, but also to evade

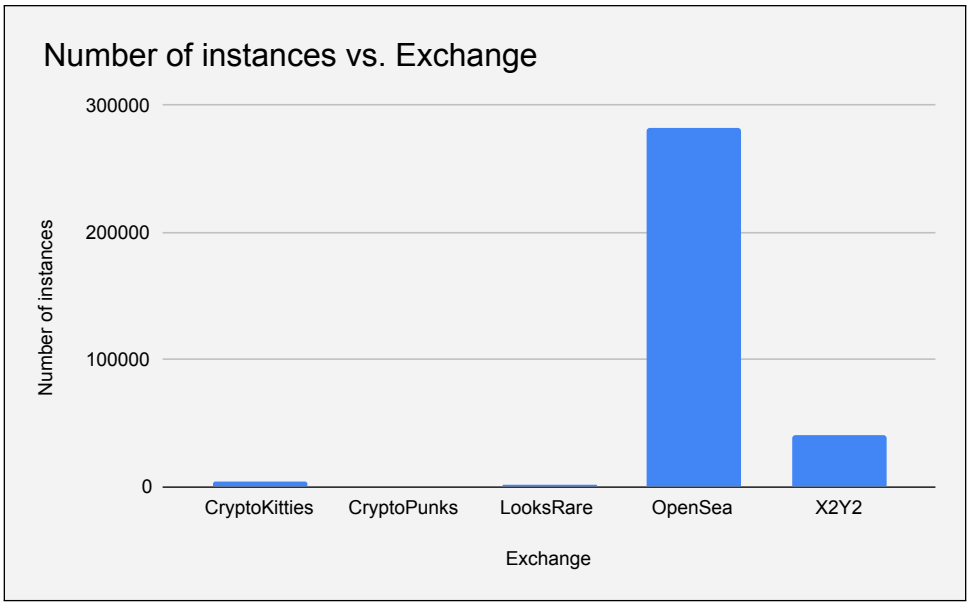

Figure 4.4: Total number of front-running instances detected per exchange

frontrunning-detection during a post-mortem analysis. Since our detection relies on the gas prices of the victim and the attack transactions being close to each other, we will fail to detect frontrunning attempts through such private mining protocols.

Results and observation. We detected 330, 238 front-running instances across 5 different NFT marketplaces. OpenSea being the largest marketplace in-terms of transaction volume, out of 330, 238 instances, 282, 747 instances occurred through OpenSea—which is around 85%. Also, out of four different types of front-running categories, we found f_{buy_buy} to be the most prevalent one. Figure 4.4 and Figure 4.5 depict the results of our findings.

Moreover, our analysis discovered that during this process the total number of NFTs bought had valuation of around $183M$ USD. Out of that buyers could sell only 129, 439 NFTs afterwards, and earned a profit of around 89M USD. However, due to the inherent risk associated with NFT buying and selling, some buyers also incurred a loss 23M USD when they sold their acquired NFTs afterwards.

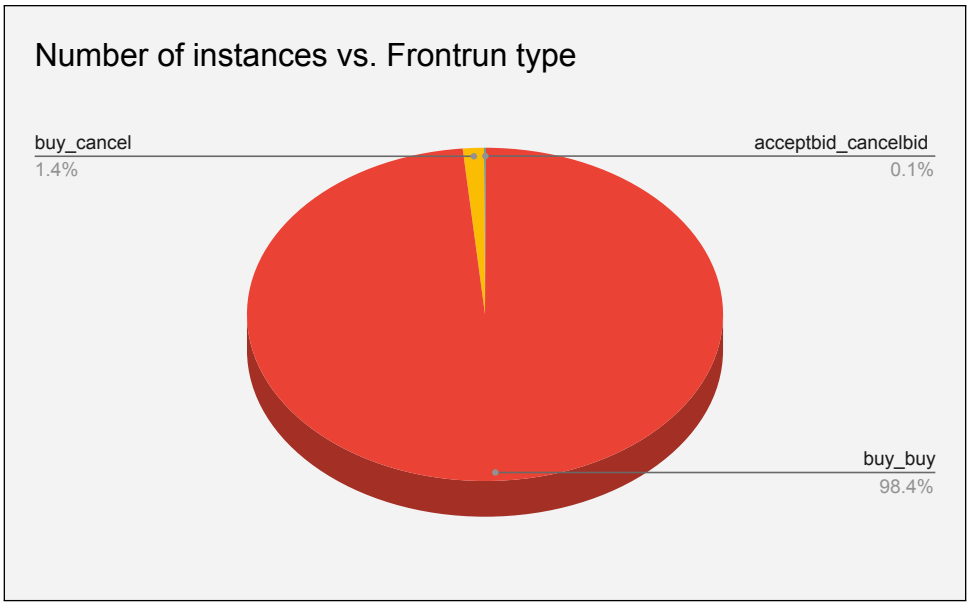

Figure 4.5: Distribution of different types of front-running instances.

Back-running.

In backrunning, the transaction sender A gets their transaction T_a ordered immediately after a target transaction T_v sent by V. In the same way as frontrunning, backrunning exploits the knowledge of the pending transactions as well. In this case, the backrunner A offers slightly lower gas price in T_a than in T_v so that their transaction gets deprioritized *w.r.t.*the target transaction during inclusion in the block. Since backrunning does not actively alter the order of the target transaction, it, alone, is not considered harmful unless paired with another frontrunning to mount a *sandwich* attack [32]. We identified the following type of opportunistic trading that leverages backrunning to acquire an NFT.

Listing–Buy. In this type of trade, the target user V is a seller, and the backrunner A is a buyer. When V submits a listing transaction T_v for a highly desirable NFT, A immediately spots this opportunity, and backruns T_v with a buy transaction T_a to acquire the NFT before others have a chance to do so. In effect, a regular user will not be able to see the NFT listed in the marketplace, and make an informed purchase decision,

Table 4.2: Back-running results.

Exchange name	Number of instances
CryptoKitties	181
CryptoPunks	130

Table 4.3: Number of back-running instances

since the item already got "sold" before even it became available for the general public. For the backrunning to be successful, it is crucial that both the seller's and the buyer's transactions get executed successfully.

Cornering.

Cornering is the act of obtaining and holding/owning enough assets, NFTs in this case, so that the holders can effectively control the market price of the items. In order to corner the market, the buyers buy a large amount of tokens as soon as a collection is launched, and hoards them until the appropriate time comes. In the regular money market, cornering the market is illegal, because it's completely unfair and manipulative. In fact, the Securities and Exchange Commission (SEC) usually monitors how many shares of a specific security are purchased by the same party.

We rely on the `transfer` logs emitted by the NFT contracts to identify when an NFT is sold, who the buyer is, and how many NFTs that buyer is holding at that point in time. Specifically, given a collection, we enumerate the the transfer logs emitted by the collection contract sequentially, ordered by block, and within each block, ordered by the log index. This ensures that the temporal relation between any two logs remains intact, *i.e.*, we process the transfer events chronologically. A `transfer` log L is of the form {`sender, receiver, tokenId`}. When a buyer purchases an NFT, the NFT is transferred to them, which, in turn, emits a `transfer` log. While iterating over the logs, we keep track of the

`tokenId` s received by the `receiver`. To understand if a purchase corners the market, we also need to know how many NFTs of that collection is in circulation at that point. Therefore, when we see a `tokenId` that we have not seen before, we consider that as a *minting* event, and update (increase) the `totalSupply` accordingly. A `transfer` event decrements and increments the number of tokens held by the `sender` (seller) as well as the `receiver` (buyer) by one, respectively. Each time someone purchases an NFT, we check what fraction f of the total number of available tokens (`totalSupply`) they own at that point. If both f and `totalSupply` are above some pre-defined threshold, that indicates that a significant portion of the tokens are under the control of that buyer. We consider that an instance of market cornering.

4.4 Profit generation

This strategy involves an experienced trader executing trades across different NFT marketplaces or a combination of NFT marketplaces and NFT/ERC-20 liquidity pools simultaneously to generate profits within a single transaction. Unlike the Acquire strategy, this approach focuses on executing transactions only if there is a potential profit remaining after accounting for transaction fees. It is not unusual for a trader to pay substantial fees to miners to expedite the trade and capitalize on a profitable opportunity before others do.

During our analysis, we found two different types of instant profit generation strategies adopted by the traders as outlined below.

4.4.1 Arbitrage

Arbitrage in crypto trading refers to the practice of taking advantage of price discrepancies for a specific cryptocurrency or asset across different exchanges or markets.

The goal is to buy the asset at a lower price on one platform and sell it at a higher price on another, profiting from the price difference. It is important to note that arbitrage opportunities in crypto trading can be short-lived and highly competitive. Traders often employ advanced trading algorithms and automated bots to quickly identify and execute trades, maximizing their chances of profiting from these fleeting opportunities. Arbitrage is a well-researched area in terms of ERC20 trading, primarily focusing on exchanges that facilitate the trading of ERC20 tokens or native currencies. The process involves capitalizing on price discrepancies between these exchanges.

On the other hand, arbitrage in NFTs is a more complex endeavor. It encompasses various protocols, exchanges, and marketplaces, such as liquidity pools and dedicated NFT marketplaces. Identifying a profitable arbitrage opportunity in the NFT space can be challenging. It often requires navigating multiple exchanges, executing different types of trade actions, and dealing with diverse currencies or tokens. In order to detect possible NFT arbritaging opportunities, we consider the following set of rules to compute profit from such a transaction.

Rule 1. For every transaction T, T should involve at-least one NFT sale. We consider the address that buys an NFT as the address of a potential trader t.

Rule 2. For every transaction T, we compute the amount of incoming and outgoing tokens (ERC-20 and NFT) and Ethers for each address.

Rule 3. At the end we check, whether t involves both ERC-20 and NFT transfers. If that satisfies, then we compute the difference of incoming Ethers into the address t and out of the address t. We denote this as profit p.

Type	Profit (ETH)	Profit (NFT)
Arbitrages	4570	-
Non-arbitrages	-	2460

Table 4.4: Total profit generated in instant profit generation

4.4.2 Non-arbitrage MEV extraction

In this strategy, the utilization of flash loan pools plays a crucial role. The trader, referred to as t_1, initiates a flash loan transaction, borrowing an amount of p_1 ETH. With the borrowed funds, t_1 proceeds to purchase an NFT of type n_1. Subsequently, by utilizing the acquired n_1 NFT, t_1 is eligible to receive an additional NFT of type n_2 as a reward. After obtaining the reward NFT, t_1 executes a sale of the initially purchased n_1 NFT. The proceeds from this sale are then used to repay the borrowed funds obtained through the flash loan. In summary, this strategy involves the strategic use of a flash loan to acquire an NFT, which in turn enables the trader to receive a different type of NFT as a reward. By selling the initially purchased NFT, the trader can generate the necessary funds to repay the flash loan and complete the transaction.

Our analysis has yielded results showcasing the detection of instant profit generation strategies, as presented in Table Table 4.4. These strategies have proven to be highly effective in generating immediate profits for the trader. Specifically, the total profit accumulated through arbitrage trades amounts to $4,570$ Ether, equivalent to approximately 13.7M USD. Furthermore, our analysis has also identified a significant number of non-arbitrage transactions where the trader has acquired NFTs as profits. In total, the trader has gained 2460 NFTs through these transactions.

Exchange name	Number of instances
CryptoKitties	25
CryptoPunks	8
LooksRare	83
OpenSea	1396
X2Y2	356

Table 4.5: Number of cancel_buy front-running instances

4.5 Loss minimization

In this particular category, the focus is on how traders employ strategies to minimize or prevent impending losses. For example in a scenario where a seller, referred to as s_1, unintentionally lists an NFT for sale at a price of p_1 ETH. However, the actual average price for such an NFT is much higher, let's say p_2 ETH. Given the significant price difference, it becomes evident that buyers will initiate transactions to take advantage of this highly lucrative opportunity. However, upon realizing the mistake, the seller, s_1, decides to cancel the listing to avoid incurring substantial losses. At this point, s_1 observes that there are already pending transactions from potential buyers looking to acquire the NFT at this mistakenly listed price. In an effort to minimize the potential loss, s_1 employs a strategy known as front-running, whereby they promptly execute their own transaction to preemptively outpace and override the pending transactions from other buyers. By front-running the existing transactions, the seller aims to minimize their potential loss by preventing the buyers from acquiring the NFT at the mistakenly listed price.

The results of our analysis to detect such front-running instances is shown in Table 4.5. Our results show that this practice is not uncommon across marketplaces, and quite intuitively OpenSea has the highest occurrence of such front-running instances.

4.6 Related work

This work focuses on the opportunistic trading of NFTs. Previous work on the high-frequency trading in the crypto market and market manipulation are closest to our research.

High-frequency trading. Arbitrage leverages the price discrepancies between cryptocurrency exchanges, and makes profit by buying cryptocurrency at a lower rate from one exchange and selling it on another. Unfortunately such opportunities may involve hundreds of tokens and exchanges, and to make the matter worse, remain open for a short amount of time. Sophisticated algorithms [120, 35, 121] have been proposed to exploit such opportunities at scale in real-time. In DeFi lending protocols, liquidation is equivalent to margin calls [150] in tradition finance. It is the process of selling off collateralized assets at a discounted price to cover outstanding debts to maintain the solvency of the protocol. In such protocols, users can borrow funds by locking up their crypto assets as collateral. However, if the value of the collateral falls below a certain threshold, or if the borrowed funds cannot be repaid, liquidation is triggered. Liquidation is open to public, but it is challenging to spot a profitable opportunity. Previous work has not only thoroughly studied the incentives and risks [151] of participating in this system, but also discovered that liquidators' efficiency has improved significantly over time, with over 70% of liquidable positions getting immediately liquidated. While typical lending protocols are collateralized, a flashloan is a DeFi lending mechanism that allows users to borrow funds without providing any collateral. Flashloans are borrowed and repaid within a single transaction—meaning that borrowers must execute the loan, and repay it in the same transaction, or the entire transaction is reverted. Since flashloan provides unconditional access to large funds, Qin *et. al.* [34] has shown how it can be leveraged to maximize arbitrage profits. High-frequency trading is even more competi-

tive due to the presence of frontrunning in Ethereum. Pending transactions and their gas bids are visible in the mempool. An attacker can be outsmarted by a frontrunner who can replay the original transaction with a higher gas price to get their transaction mined early and pocket the profit [30]. FlashBoys [31] has demonstrated that arbitrage bots often engage in a reactive bidding against each other until the system attains an equilibrium. A2MM [124] proposes a DEX design that mitigates the risk of predatory frontrunning attacks. Sandwich attacks, as the name suggests, is about extracting value from a victim transaction by both frontrunning and backrunning it at the same time. Quite shockingly, such attacks have been shown [32] to generate a daily revenue of over several thousand US dollars just from Uniswap, a popular DEX. Blockchain extractable value (BEV) is an umbrella term that refers to all those different ways, such as sandwich, liquidation, and arbitrage, to make illicit or harmful profits from the blockchain. A body of work [33, 122, 123] has studied the prevalence of such trades, their impact, and the revenue they generate in depth.

Existing work focuses on native cryptocurrency and ERC-20 token markets. To our knowledge, we are the first to explore the presence of similar opportunistic trades in the NFT space, which brings unique challenges associated to speculative pricing, diverse trade actions, and real-time detection of profitability. In this work, not only do we show how some of the trades we have seen before are manifested differently in the NFT market, but also we uncover instances of NFT-specific, novel opportunistic trading techniques.

Manipulation of token markets. An NFT rug pull refers to a fraudulent practice where the creator or seller of an NFT abruptly abandons the project, leaving investors with worthless or significantly devalued tokens. Previous work has shown the prevalence of rug pulls [125, 126, 127], and proposed models for detection based on early symptoms. Analysis models [128, 129, 130, 131] have been proposed to detect ponzi schemes, which is is a fraudulent investment scheme where the operator promises high and consistent returns

to participants by using funds from new investors to pay returns to earlier investors. Pump-and-dump [132, 133, 134] is a manipulative practice, where attackers artificially inflate the price of a particular cryptocurrency (pump), and then quickly sell off their holdings at the inflated price (dump), leaving other investors with significant losses. Other forms of market manipulations, like, wash trading, shill bidding, and bid shielding have also heavily been researched [135, 136, 137, 138, 139, 140, 141].

4.7 Conclusion

In this chapter, we shed light on the evolving landscape of cryptocurrency (CC) trading with respect to fungible/non-fungible tokens (NFTs). While the financial ecosystem built on blockchain technology is expected to be fair for all participants, the reality is that sophisticated actors leverage their domain knowledge and market inefficiencies to gain strategic advantages, often extracting value from trades that are not accessible to others. The under-regulated nature of the cryptocurrency market exacerbates these issues, further amplifying concerns related to fairness and transparency. While previous research has explored various aspects of unfairness in CC trading, the economic intricacies of NFT trades remain largely unexplored, making this study particularly significant. The findings reveal several noteworthy observations. Firstly, sophisticated actors employ automated, high-frequency NFT trading strategies, often exhibiting malicious, illicit, or unfair behavior. Secondly, many strategies applicable to traditional CC or fungible token trading also apply to NFTs. Lastly, the unique nature of the NFT market creates specific opportunities for threat actors to exploit. In conclusion, this study contributes to the growing body of knowledge on CC trading by examining the dynamics of the NFT market and highlighting the presence of unfair practices.

Chapter 5

Conclusion and future research

In this dissertation, we presented novel approaches to study the security and usability issues of event-driven applications with respect to different types of event-driven applications, such as, Android apps, smart contract, *etc.*

In particular, we proposed COLUMBUS, a callback-driven Android app testing technique that improves over the state-of-the-art in three aspects: **(i)** systematically identifying the callbacks present in an app, **(ii)** inferring coverage maximizing primitive arguments, while generating object arguments in an Android API-agnostic manner, and **(iii)** providing *data dependency* and *crash-guidance* as 'feedback' to increase the probability of triggering uninitialized data related crashes, and preventing the tool from rediscovering the same bugs, respectively. In our evaluation, COLUMBUS outperformed state-of-the-art model-driven, checkpoint-based, and callback-driven testing tools both in terms of crashes and coverage.

Next, we propose SAILFISH, a scalable hybrid tool for automatically identifying SI bugs in smart contracts. SAILFISH leverages lightweight exploration phase followed by symbolic evaluation aided by our novel VSA. On the ETHERSCAN dataset, SAILFISH significantly outperforms state of the art analyzers in terms of precision, and performance,

identifying 47 previously unknown vulnerable (and exploitable) contracts.

Also, we analyze how sophisticated actors employ automated, high-frequency NFT trading strategies, often exhibiting malicious, illicit, or unfair behavior. Many strategies applicable to traditional cryptocurrency or fungible token trading also apply to NFTs. The nature of the NFT market creates unique opportunities for threat actors. Building upon these insights, we delve into three broad classes of "opportunistic" trading strategies: acquire (buy-and-hold), instant profit generation, and loss minimization. By studying and understanding these trading strategies, we contribute to shedding light on the dynamics and challenges within the NFT market. It emphasizes the need for further exploration, regulation, and measures to promote fairness and integrity in NFT trading.

Lastly, this thesis serves as a valuable foundation for future research directions in the field. The following areas can be explored to further advance the understanding and improvement of security and usability in event-driven applications.

A promising research direction involves improving Sailfish's symbolic execution engine to handle live transactions. This includes devising methods to handle dynamic data and interactions in real-time, ensuring compatibility with dynamic transactions, and addressing scalability concerns to enable symbolic execution on live blockchain transactions.

Another research opportunity lies in enhancing the object argument generation module of Columbus. This entails improving the initialization of nested object arguments, resulting in increased precision and reduced false positives. Research efforts can focus on tackling the complexities associated with object structures and enhancing the accuracy of identifying coverage-maximizing primitive arguments while generating object arguments in an Android API-agnostic manner.

By pursuing these research directions, further advancements can be made in the fields of security and usability in event-driven applications.

Bibliography

[1] "Understanding the dao attack." https://tinyurl.com/yc3o8ffk/, 2017.

[2] "On the parity wallet multisig hack." https://tinyurl.com/yca83zsg/, 2017.

[3] "Governmental's 1100 eth payout is stuck because it uses too much gas.." https://tinyurl.com/y83dn2yf/, 2016. [accessed 01/09/2019].

[4] N. Atzei, M. Bartoletti, and T. Cimoli, *A survey of attacks on ethereum smart contracts (sok)*, in *Principles of Security and Trust - 6th International Conference, POST*, 2017.

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

[6] "Exploiting uniswap: from reentrancy to actual profit.." https://blog.openzeppelin.com/exploiting-uniswap-from-reentrancy-to-actual-profit/, 2019.

[7] "Reentering the reentrancy bug: Disclosing burgerswap's vulnerability.." https://www.zengo.com/burgerswap-vulnerability/. accessed 10/22/2020].

[8] "The reentrancy strikes again - the case of lendf.me." https://valid.network/post/the-reentrancy-strikes-again-the-case-of-lendf-me.

[9] "Android statistics." https://www.businessofapps.com/data/android-statistics.

[10] "Android app release statistics." https://www.statista.com/statistics/1020956/android-app-releases-worldwide/.

[11] "Etherscan." https://etherscan.io/, 2018. [accessed 01/09/2019].

[12] "Real estate business integrates smart contracts." https://tinyurl.com/yawrkfpx/. [accessed 01/09/2019].

[13] "Smart contracts for shipping offer shortcut." https://tinyurl.com/yavel7xe/.

[14] "Monkey." `http://developer.android.com/tools/help/monkey.html`.

[15] "Uiautomator." `https://developer.android.com/training/testing/other-components/ui-automator`.

[16] B. Y and B. D, *Automated model-based android gui testing using multi-level gui comparison criteria*, in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[17] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, *Guided, stochastic model-based GUI testing of android apps*, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017.

[18] W. Choi, G. Necula, and K. Sen, *Guided gui testing of android apps with minimal restart and approximate learning*, in *Proc. of OOPSLA*, vol. 2013, 2013.

[19] W. Yang, M. R. Prasad, and T. Xie, *A grey-box approach for automated gui-model generation of mobile applications*, in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, 2013.

[20] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, *Practical gui testing of android applications via model abstraction and refinement*, in *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[21] W. Song, X. Qian, and J. Huang, *Ehbdroid: Beyond gui testing for android applications*, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2017.

[22] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, *Edgeminer: Automatically detecting implicit control flow transitions through the android framework.*, in *NDSS*, 2015.

[23] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, *TXSPECTOR: Uncovering attacks in ethereum from transactions*, in *29th USENIX Security Symposium (USENIX Security)*, 2020.

[24] M. Rodler, W. Li, G. O. Karame, and L. Davi, *Sereum: Protecting existing smart contracts against re-entrancy attacks*, in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[25] 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.

[26] 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.

[27] "Mythril." `https://github.com/ConsenSys/mythril`. [accessed 07/27/2020].

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

[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] S. Eskandari, S. Moosavi, and J. Clark, *Sok: Transparent dishonesty: Front-running attacks on blockchain*, 2020.

[31] 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 *SP*, 2020.

[32] L. Zhou, K. Qin, C. F. Torres, D. Le, and A. Gervais, *High-frequency trading on decentralized on-chain exchanges*, in *SP*, 2020.

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

[34] K. Qin, L. Zhou, B. Livshits, and A. Gervais, *Attacking the defi ecosystem with flash loans for fun and profit*, 2021.

[35] 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 *IEEE SP*, 2021.

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

[37] 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).

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

[39] "Source code of columbus." `https://github.com/ucsb-seclab/columbus`.

[40] S. Roy Choudhary, A. Gorla, and A. Orso, *Automated test input generation for android: Are we there yet? (e)*, 11, 2015.

[41] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, *Time-travel testing of android apps*, in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 481–492, IEEE, 2020.

[42] J. Palsberg and M. I. Schwartzbach, *Object-oriented type inference*, in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1991.

[43] M. Reif, F. Kübler, M. Eichberg, and M. Mezini, *Systematic evaluation of the unsoundness of call graph construction algorithms for java*, in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018.

[44] K. Mao, M. Harman, and Y. Jia, *Sapienz: Multi-objective automated testing for android applications*, in *Proceedings of the International Symposium on Software Testing and Analysis*, 2016.

[45] A. Machiry, R. Tahiliani, and M. Naik, *Dynodroid: An input generation system for android apps*, in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[46] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, *Reducing combinatorics in gui testing of android applications*, in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[47] S. Anand, M. Naik, M. J. Harrold, and H. Yang, *Automated concolic testing of smartphone apps*, in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12*, p. 59, 2012.

[48] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, *Using gui ripping for automated testing of android applications*, in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[49] W. Choi, G. Necula, and K. Sen, *Guided gui testing of android apps with minimal restart and approximate learning*, in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[50] W. Yang, M. R. Prasad, and T. Xie, *A grey-box approach for automated gui-model generation of mobile applications*, in *Fundamental Approaches to Software Engineering* (V. Cortellessa and D. Varró, eds.), 2013.

[51] R. Mahmood, N. Mirzaei, and S. Malek, *Evodroid: segmented evolutionary testing of android apps*, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, 2014.

[52] "Google play store." `https://play.google.com/`.

[53] "Emma, a java code coverage tool." http://emma.sourceforge.net.

[54] "Logcat." https://developer.android.com/studio/command-line/logcat.

[55] "angr, binary analysis framework." https://angr.io.

[56] "Frida, dynamic instrumentation toolkit." https://frida.re.

[57] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, *Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps*, in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pp. 204–217, 2014.

[58] L. J. White and H. Almezen, *Generating test cases for GUI responsibilities using complete interaction sequences*, in *11th International Symposium on Software Reliability Engineering (ISSRE 2000)*, 2000.

[59] X. Yuan and A. M. Memon, *Generating event sequence-based test cases using GUI runtime state feedback*, *IEEE Trans. Software Eng.* (2010).

[60] H. van der Merwe, B. van der Merwe, and W. Visser, *Verifying android applications using java pathfinder*, *ACM SIGSOFT Softw. Eng. Notes* (2012) 1–5.

[61] Y. Li, Z. Yang, Y. Guo, and X. Chen, *Droidbot: a lightweight ui-guided test input generator for android*, in *ICSE 2017*.

[62] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, *Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps*, *IEEE Trans. Software Eng.* (2022) 1115–1137.

[63] N. P. B. Jr., J. Hotzkow, and A. Zeller, *Droidmate-2: a platform for android test generation*, in *ASE*, 2018.

[64] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, *Mobiguitar: Automated model-based testing of mobile apps*, *IEEE Softw.* (2015) 53–59.

[65] A. M. Memon, I. Banerjee, and A. Nagarajan, *GUI ripping: Reverse engineering of graphical user interfaces for testing*, in *WCRE 2003*.

[66] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, *Triggerscope: Towards detecting logic bombs in android applications*, in *2016 IEEE symposium on security and privacy (SP)*, pp. 377–396, IEEE, 2016.

[67] C. S. Jensen, M. R. Prasad, and A. Møller, *Automated testing with targeted event sequence generation*, in *International Symposium on Software Testing and Analysis, ISSTA '13*, 2013.

[68] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood, *Testing android apps through symbolic execution*, *ACM SIGSOFT Softw. Eng. Notes* (2012) 1–5.

[69] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, *Android testing via synthetic symbolic execution*, in *ASE 2018*, 2018.

[70] T. Azim and I. Neamtiu, *Targeted and depth-first exploration for systematic testing of android apps*, in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*, 2013.

[71] D. Lai and J. Rubin, *Goal-driven exploration for android applications*, in *ASE*, 2019.

[72] W. Yang, M. R. Prasad, and T. Xie, *A grey-box approach for automated gui-model generation of mobile applications*, in *Fundamental Approaches to Software Engineering (FASE) 2013*, 2013.

[73] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, *Multiple-entry testing of android applications by constructing activity launching contexts*, in *ICSE*, 2020.

[74] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie, *Improving automated GUI exploration of android apps via static dependency analysis*, in *ICSME*, 2020.

[75] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, *Combodroid: generating high-quality test inputs for android apps via use case combinations*, in *ICSE*, 2020.

[76] A. Sadeghi, R. Jabbarvand, and S. Malek, *Patdroid: permission-aware GUI testing of android*, in *ESEC/FSE*, 2017.

[77] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon, *Lightweight static analysis for GUI testing*, in *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012*, pp. 301–310, 2012.

[78] M. Y. Wong and D. Lie, *Driving execution of target paths in android applications with (a) car*, in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2022.

[79] Y. Feng, E. Torlak, and R. Bodík, *Summary-based symbolic evaluation for smart contracts*, in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*, 2020.

[80] P. Godefroid, *Compositional dynamic test generation*, in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2007.

[81] S. Anand, P. Godefroid, and N. Tillmann, *Demand-driven compositional symbolic execution*, in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, 2008.

[82] S. Wang, C. Zhang, and Z. Su, *Detecting nondeterministic payment bugs in ethereum smart contracts*, *Proc. ACM Program. Lang.* **3** (2019), no. OOPSLA.

[83] "Swc 114 - transaction order dependence attack.." `https://swcregistry.io/docs/SWC-114`. [accessed 04/26/2020].

[84] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, *Vandal: A scalable security analysis framework for smart contracts*, 2018.

[85] "Sereum repository." `https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns/`, 2019.

[86] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, *Exploiting the laws of order in smart contracts*, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[87] F. M. Quintao Pereira, R. E. Rodrigues, and V. H. Sperle Campos, *A fast and low-overhead technique to secure programs against integer overflows*, in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[88] E. Torlak and R. Bodík, *A lightweight symbolic virtual machine for solver-aided host languages*, in *Proc. Conference on Programming Language Design and Implementation*, 2014.

[89] J. Feist, G. Grieco, and A. Groce, *Slither: A static analysis framework for smart contracts*, in *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019.

[90] "Celery - distributed task queue." `https://celeryproject.org`, 2020.

[91] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, *Smartcheck: Static analysis of ethereum smart contracts*, in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

[92] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, *Ethor: Practical and provably sound static analysis of ethereum smart contracts*, in *Proc. Conference on Computer and Communications Security*, 2020.

[93] 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.

[94] "Cream finance post mortem: Amp exploit." https://medium.com/cream-finance/c-r-e-a-m-finance-post-mortem-amp-exploit-6ceb20a630c5.

[95] J. Jiao, S. Kan, S. Lin, D. Sanan, Y. Liu, and J. Sun, *Semantic understanding of smart contracts: Executable operational semantics of solidity*, in *IEEE Symposium on Security and Privacy (SP)*, 2020.

[96] "Rattle: Evm static analysis framework." https://github.com/crytic/rattle.

[97] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, *Gigahorse: Thorough, declarative decompilation of smart contracts*, in *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[98] "Panoramix decompiler." https://github.com/palkeo/panoramix.

[99] S. Lagouvardos, N. Grech, I. Tsatiris, and Y. Smaragdakis, *Precise static modeling of ethereum memory*, .

[100] J. Frank, C. Aschermann, and T. Holz, *ETHBMC: A bounded model checker for smart contracts*, in *29th USENIX Security Symposium (USENIX Security)*, 2020.

[101] Y. Feng, E. Torlak, and R. Bodik, *Precise attack synthesis for smart contracts*, *arXiv preprint arXiv:1902.06067* (2019).

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

[103] J. Krupp and C. Rossow, *teether: Gnawing at ethereum to automatically exploit smart contracts*, in *Proc. USENIX Security Symposium*, 2018.

[104] T. Chen, R. Cao, T. Li, X. Luo, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, X. Lin, and X. Zhang, *Soda: A generic online detection framework for smart contracts*, in *Proc. The Network and Distributed System Security Symposium*, 2020.

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

[106] V. Wüstholz and M. Christakis, *Harvey: A greybox fuzzer for smart contracts*, *ArXiv* **abs/1905.06944** (2019).

[107] V. Wüstholz and M. Christakis, *Targeted greybox fuzzing with static lookahead analysis*, .

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

[109] 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.

[110] F. Brown, D. Stefan, and D. Engler, *Sys: A static/symbolic tool for finding good bugs in good (browser) code*, in *29th USENIX Security Symposium (USENIX Security)*, USENIX Association, 2020.

[111] H. Cui, G. Hu, J. Wu, and J. Yang, *Verifying systems rules using rule-directed symbolic execution*, *SIGARCH Comput. Archit. News* (2013).

[112] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, *Precise and scalable detection of double-fetch bugs in os kernels*, in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[113] D. Babić, L. Martignoni, S. McCamant, and D. Song, *Statically-directed dynamic automated test generation*, in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011.

[114] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet, *Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free*, in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 2016.

[115] A. Y. Gerasimov, *Directed dynamic symbolic execution for static analysis warnings confirmation*, *Program. Comput. Softw.* (2018).

[116] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, *Assertion guided symbolic execution of multithreaded programs*, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[117] S. Guo, M. Kusano, and C. Wang, *Conc-ise: Incremental symbolic execution of concurrent software*, in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[118] E. Cecchetti, S. Yao, H. Ni, and A. C. Myers, *Compositional security for reentrant applications*, in *IEEE Symposium on Security and Privacy (SP)*, 2021.

[119] D. Perez and B. Livshits, *Smart contract vulnerabilities: Vulnerable does not imply exploited*, in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[120] R. McLaughlin, C. Kruegel, and G. Vigna, *A large scale study of the ethereum arbitrage ecosystem*, in *Proc. USENIX Security Symposium*, 2023.

[121] Y. Wang, Y. Chen, H. Wu, L. Zhou, S. Deng, and R. Wattenhofer, *Cyclic arbitrage in decentralized exchanges*, in *Proc. Web Conference*, 2022.

[122] J. Piet, J. Fairoze, and N. Weaver, *Extracting godl [sic] from the salt mines: Ethereum miners extracting value*, 2022.

[123] M. Bartoletti, J. H.-y. Chiang, and A. Lluch Lafuente, *Maximizing extractable value from automated market makers*, in *Proc. Financial Cryptography and Data Security* (I. Eyal and J. Garay, eds.), 2022.

[124] L. Zhou, K. Qin, and A. Gervais, *A2mm: Mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges*, 2021.

[125] S. S. Roy, D. Das, P. Bose, C. Kruegel, G. Vigna, and S. Nilizadeh, *Demystifying nft promotion and phishing scams*, 2023.

[126] J. Huang, N. He, K. Ma, J. Xiao, and H. Wang, *A deep dive into nft rug pulls*, 2023.

[127] T. Sharma, R. Agarwal, and S. K. Shukla, *Understanding rug pulls: An in-depth behavioral analysis of fraudulent nft creators*, 2023.

[128] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, *Dissecting ponzi schemes on ethereum: Identification, analysis, and impact*, *Future Generation Computer Systems* **102** (2020).

[129] W. Chen, Z. Zheng, E. C.-H. Ngai, P. Zheng, and Y. Zhou, *Exploiting blockchain data to detect smart ponzi schemes on ethereum*, *IEEE Access* **7** (2019).

[130] W. Chen, X. Li, Y. Sui, N. He, H. Wang, L. Wu, and X. Luo, *Sadponzi: Detecting and characterizing ponzi schemes in ethereum smart contracts*, *SIGMETRICS Perform. Eval. Rev.* **5** (2021), no. 2.

[131] T. Kell, H. Yousaf, S. Allen, S. Meiklejohn, and A. Juels, *Forsage: Anatomy of a smart-contract pyramid scheme*, in *Proc. Financial Cryptography and Data Security*, 2021.

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

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

[134] J. Xu and B. Livshits, *The anatomy of a cryptocurrency pump-and-dump scheme*, 2019.

[135] D. Das, P. Bose, N. Ruaro, C. Kruegel, and G. Vigna, *Understanding security issues in the nft ecosystem*, in *Proc. Conference on Computer and Communications Security*, 2022.

[136] V. von Wachter, J. R. Jensen, F. Regner, and O. Ross, *Nft wash trading: Quantifying suspicious behaviour in nft markets*, 2022.

[137] M. L. Morgia, A. Mei, A. M. Mongardini, and E. N. Nemmi, *A game of nfts: Characterizing nft wash trading in the ethereum blockchain*, 2023.

[138] G. Bonifazi, F. Cauteruccio, E. Corradini, M. Marchetti, D. Montella, S. Scarponi, D. Ursino, and L. Virgili, *Performing wash trading on nfts: Is the game worth the candle?*, *Big Data and Cognitive Computing* **7** (2023), no. 1.

[139] S. Serneels, *Detecting wash trading for nonfungible tokens*, *Finance Research Letters* **52** (2023).

[140] D. Liu, F. Piccoli, K. Chen, A. Tang, and V. Fang, *Nft wash trading detection*, 2023.

[141] X. Wen, Y. Wang, X. Yue, F. Zhu, and M. Zhu, *Nftdisk: Visual detection of wash trading in nft markets*, in *Proc. International Conference on Human Factors in Computing Systems*, 2023.

[142] I. Bogatyy, "How we made $100k trading cryptokitties." `https://medium.com/@ivanbogatyy/how-we-made-100k-trading-cryptokitties-2d69aebe715b`.

[143] "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`.

[144] "Nftx." `https://nftx.io`.

[145] 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 *NDSS*, 2021.

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

[147] 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 *Proc. USENIX Security Symposium*, 2023.

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

[149] "Flashbot mev boost." `https://docs.flashbots.net/flashbots-mev-boost/introduction`.

[150] "Margin call: What it is and how to meet one with examples." `https://www.investopedia.com/terms/m/margincall.asp`.

[151] K. Qin, L. Zhou, P. Gamito, P. Jovanovic, and A. Gervais, *An empirical study of defi liquidations: Incentives, risks, and instabilities*, in *Proc. ACM Internet Measurement Conference*, 2021.