

UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

Uncovering Software Threats Through Retrospective Security Analysis

Permalink

<https://escholarship.org/uc/item/55g1753d>

ISBN

9798186381327

Author

Meng, Dongyu

Publication Date

2026-05-19

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Santa Barbara

Uncovering Software Threats Through Retrospective Security Analysis

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

Doctor of Philosophy

in

Computer Science

by

Dongyu Meng

Committee in Charge:

Professor Giovanni Vigna, Co-Chair

Professor Christopher Kruegel, Co-Chair

Professor Wenbo Guo

June 2026

The Dissertation of
Dongyu Meng is approved:

Wenbo Guo

Christopher Kruegel, Committee Co-Chair

Giovanni Vigna, Committee Co-Chair

April 2026

Uncovering Software Threats Through
Retrospective Security Analysis

Copyright © 2026

by

Dongyu Meng

Acknowledgements

This PhD journey was longer and stranger than I imagined. Luckily, I was not alone.

My family has been with me the whole way through. My parents supported my decision to start this adventure in the US, though it inevitably meant I'd rarely be home. They never complained. Long before this PhD, they made hard life choices to make sure that I could get the best education possible. I wouldn't be here without their sacrifice. My parents-in-law took me in and supported both me and my wife through uncertain days. And to my grandparents, who are no longer with us, I miss you, and I wish you could have seen this.

To my advisors Chris and Giovanni: thank you for sticking with me. For the guidance, yes, but also for the patience, for not giving up when things went sideways, and for giving me the room to find my way through. To my fellow Seclabbers, my friends and war buddies in equal measure: the lab felt like home because of each and every one of you.

And finally, to my wife, Jingyi. None of this happens without you. Thank you for being here with me. I love you.

Curriculum Vitæ

Dongyu Meng

Education

2018 – 2026	PhD in Computer Science University of California, Santa Barbara
2015 – 2018	MS in Communication and Information Systems University of Chinese Academy of Sciences, Shanghai, China
2011 – 2015	BE in Automation Tsinghua University, Beijing, China

Publications

1. **Dongyu Meng**, Christopher Kruegel, and Giovanni Vigna. “Dissonance: Detecting Behavioral Regressions in Smart Contract Upgrades”. under submission.
2. **Dongyu Meng***, Fabio Gritti*, Robert McLaughlin, Nicola Ruaro, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. “HOUSTON: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols.” In *33rd Annual Network and Distributed System Security Symposium (NDSS 2026)*. 2026.
3. **Dongyu Meng**, Michele Guerriero, Aravind Machiry, Hojjat Aghakhani, Priyanka Bose, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. “Bran: Reduce Vulnerability

- Search Space in Large Open Source Repositories by Learning Bug Symptoms.” In *16th ACM Asia Conference on Computer and Communications Security (AsiaCCS 2021)*. 2021.
4. **Dongyu Meng** and Hao Chen. “MagNet: A Two-Pronged Defense against Adversarial Examples.” In *24th ACM Conference on Computer and Communications Security (CCS 2017)*. 2017.
 5. Robert McLaughlin, **Dongyu Meng**, Christopher Kruegel, and Giovanni Vigna. “No Talking Spring: When Price Oracles Fall into Disrepair”. under submission.
 6. Zihao Su, Kyle Zeng, **Dongyu Meng**, Yigitcan Kaya, Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. “Characterizing and Detecting Harassment in Social Virtual Reality”. under submission.
 7. Nicola Ruaro, Fabio Gritti, **Dongyu Meng**, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. “Approve Once, Regret Forever: On the Exploitation of Ethereum’s Approve-TransferFrom Ecosystem.” In *34th USENIX Security Symposium (USENIX Security 2025)*. 2025.
 8. Nicola Ruaro, Fabio Gritti, Robert McLaughlin, **Dongyu Meng**, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. “A History of Greed: Practical Symbolic Execution for Ethereum Smart Contracts.” In *22nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2025)*. 2025.
 9. Hojjat Aghakhani, **Dongyu Meng**, Yu-Xiang Wang, Christopher Kruegel, and Giovanni Vigna. “Bullseye Polytope: A Scalable Clean-Label Poisoning Attack with Improved Trans-

ferability.” In *6th IEEE European Symposium on Security and Privacy (EuroS&P 2021)*. 2021.

10. Yuanyi Wu, **Dongyu Meng**, and Hao Chen. “Evaluating Private Modes in Desktop and Mobile Browsers and Their Resistance to Fingerprinting.” In *2017 IEEE Conference on Communications and Network Security (CNS 2017)*. 2017.

Abstract

Uncovering Software Threats Through Retrospective Security Analysis

by

Dongyu Meng

Modern software systems are growing in complexity at a pace that surpasses our ability to secure them. Vulnerabilities hide in millions of lines of code in large open source software repositories, in the intricate business logic of financial protocols, and in the silent behavioral shifts introduced by software updates. At the heart of the problem is a deceptively simple question: how do we recognize when software is behaving in a way that it should not? Answering this question requires a reference for what “normal” looks like, and this dissertation argues that such a reference already exists, embedded in the history of software itself. The development process of a codebase, the execution traces of a deployed protocol, and the version history of an upgradeable contract all encode rich information about expected behavior. By mining these retrospective artifacts intelligently, we can build practical, automated tools that identify security threats across a wide range of software domains, from operating system kernels to blockchain-based financial applications.

Throughout my Ph.D., I designed and built three systems that instantiate this thesis. First, I present BRAN, a vulnerability prioritization system that combines code complexity

metrics with repository metadata to predict which functions in a large codebase are most likely to harbor undiscovered vulnerabilities, reducing the search space for both manual auditing and automated testing. Second, I present HOUSTON, a real-time anomaly detection system for Ethereum DeFi protocols that builds behavioral models from historical transaction execution traces and flags live transactions that deviate from established norms, detecting attacks without relying on known vulnerability patterns. Third, I present DISSONANCE, a differential analysis system that uses symbolic execution to compare successive versions of upgradeable smart contracts, identifying behavioral deviations that may signal malicious or unintended changes. Together, these systems demonstrate that retrospective software artifacts are rich sources of security signal that can be harnessed to detect threats.

Contents

Acknowledgements	iv
Curriculum Vitæ	v
Abstract	viii
List of Figures	xiii
List of Tables	xiv
1 Introduction	1
1.1 Background	3
1.2 Contributions and Dissertation Outline	5
2 Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms	9
2.1 Introduction	9
2.2 Approach	14
2.2.1 Code Metric Features	15
2.2.2 Repository Metadata Features	16
2.2.3 Dataset	22
2.3 Implementation	23
2.3.1 Inputs Processing	23
2.3.2 Learning Bug Symptoms	26
2.4 Evaluation	27
2.4.1 Datasets	29
2.4.2 Prediction Capabilities	31

2.4.3	Comparison with Existing Tools	34
2.4.4	Training Set Balance	39
2.4.5	Bug Symptoms Knowledge Transfer	40
2.4.6	BRAN-guided Fuzzing	43
2.5	Discussion and Limitation	45
2.6	Related Work	47
2.7	Conclusion	49
3	Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols	52
3.1	Introduction	52
3.2	Background	57
3.2.1	Smart Contracts & DeFi	57
3.2.2	Anomaly Detection	59
3.3	Motivation	59
3.4	Running Example: TicketMonster	63
3.5	System Design	65
3.5.1	Contract Processor	68
3.5.2	Transaction Processor	70
3.5.3	Protocol Interaction Analysis	71
3.5.4	Invariant Analysis	74
3.5.5	Anomaly Detection	79
3.5.6	Continuous Updates	81
3.6	Evaluation	82
3.6.1	DeFi Attacks Dataset	82
3.6.2	Experimental Setup	83
3.6.3	Historical Evaluation	84
3.6.4	Comparison with Existing Systems	91
3.6.5	HOUSTON Live Performance Evaluation	95
3.6.6	HOUSTON for other EVM-compatible chains	99
3.7	Discussion and Limitations	100
3.8	Related Work	102
3.9	Conclusions	105
3.10	Appendix	106
4	Dissonance: Detecting Behavioral Regressions in Smart Contract Upgrades	122
4.1	Introduction	122
4.2	Background	127

4.2.1	Ethereum Smart Contracts	127
4.2.2	Upgradeable Smart Contracts and the Proxy Pattern	128
4.2.3	Symbolic Execution	129
4.3	Approach	131
4.3.1	Function Classification and Filtering	132
4.3.2	Behavioral Profile Extraction	134
4.3.3	Storage and Call Comparison	136
4.3.4	Path Constraint Comparison	139
4.3.5	Detection	140
4.3.6	Static Analysis for Newly Added Functions	145
4.4	Evaluation	148
4.4.1	Datasets	148
4.4.2	Ground-Truth Validation	149
4.4.3	Discovery Run	151
4.4.4	Triage and Precision	153
4.5	Discussion	155
4.5.1	Limitations	155
4.5.2	Specific False-Positive Patterns	156
4.6	Related Work	157
4.7	Conclusion	161
5	Conclusions and Future Work	166
5.1	Future Directions	167
	Bibliography	168

List of Figures

2.1	BRAN’s performance in same-repository setting.	32
2.2	BRAN’s performance in same-repository setting (high ranking functions). .	32
2.3	Correlation between model performance and vulnerable-to-non-vulnerable function ratio.	40
2.4	BRAN’s performance in cross-repository setting.	42
3.1	<i>TicketMonster</i> ’s Web2 frontend.	63
3.2	HOUSTON architecture overview.	65
3.3	Protocol Interaction Analysis demonstration.	75
3.4	Cumulative distribution of false positive rates.	91
3.5	Overview of the correlation of false positive rates with the total number of transactions available per protocol.	92
3.6	Cumulative distribution of false positive rates of the <i>Interaction Model</i> with two different configurations.	120
3.7	Total number of false positives of the <i>Invariant Model</i> across all protocols with different hyperparameter setups.	121
4.1	Overview of the DISSONANCE analysis pipeline.	132

List of Tables

2.1	Features computed by BRAN grouped by category.	18
2.2	Number of functions in each dataset.	31
2.3	Results of guided fuzzing.	51
3.1	Example of the KECCAK256 look-up table collected during the transaction tracing.	106
3.2	Overview of DeFi Attacks Dataset.	108
3.3	HOUSTON execution time.	112
3.4	Transaction load for HOUSTON during live experiments.	112
3.5	Evaluation results on high TVL protocols with <i>no known</i> history of exploitation.	113
3.6	HOUSTON historical evaluation results.	116
4.1	DISSONANCE ground-truth incident dataset.	162
4.2	DISSONANCE ground-truth detection results.	163
4.3	Discovery-run findings by detector.	164
4.4	DISSONANCE reports triage results by detector and track.	165

Chapter 1

Introduction

Modern software has grown dramatically in both scale and diversity. The Linux kernel alone exceeds 30 million lines of code contributed by thousands of developers [122], and a single Ethereum DeFi protocol may coordinate dozens of interacting smart contracts managing millions of dollars in assets [37]. Each new domain, from operating system internals to decentralized finance, introduces its own conventions, trust boundaries, and failure modes.

These highly complex systems underpin critical infrastructure, financial markets, and the daily lives of billions of people, and their security failures carry consequences far beyond the software itself. A memory corruption bug in a widely deployed library can be weaponized into ransomware that shuts down hospitals across continents [26]. A logic flaw in a smart contract can be exploited in seconds to drain hundreds of millions of dollars, with no possibility of reversal [32, 150]. A seemingly innocuous contract upgrade

can silently introduce a bug that undermines assumptions an entire ecosystem was built on [33]. In each case, security threats may stem from the software transitioning into a state that its developers never designed for: a bug pushes a value outside its intended range, an attacker chains operations in an order no one considered, or an update quietly relaxes a check that downstream code relied on. Securing these systems requires methods that can anticipate when and where these threats are likely to emerge.

A long line of security research has recognized that threats manifest as *anomalies*: deviations from some expected pattern of behavior [21, 38]. Intrusion detection systems flag unusual system-call sequences [51], malware detectors identify statistical outliers in program behavior [165], and vulnerability predictors look for code whose characteristics diverge from those of known-good modules [151]. The principle is well understood, but the practical challenge has always been where to obtain a robust reference for normal behavior in the first place. Formal specification is the textbook answer: write down exactly what the program should do and verify the code against it. In practice, this is rarely feasible. Real systems are too large to fully specify, change too quickly for any specification to keep up, and operate in domains where “correct” is itself a moving target shaped by users, markets, and regulations [182]. Pattern-based detectors and hand-written rules sidestep the specification problem by encoding known bad behaviors, but they pay for it in coverage: they only catch what someone already thought to look for [203].

This dissertation argues for a different source of behavioral reference. Software does not appear fully formed. It is built by teams over months or years, executed by real users under real conditions, and revised through successive versions that each inherit assumptions from their predecessors. Each of these processes produces artifacts that encode, implicitly but richly, what “normal” looks like for that particular system. We call these *retrospective artifacts*: the traces left behind by software development, deployment, and evolution. None of them is a specification, but collectively, they describe a working notion of expected behavior that is grounded in what the software, or the development of the software, has actually done, rather than in what someone hoped it would do. The central claim of this dissertation is that mining these artifacts yields practical and effective security analyses.

1.1 Background

Before describing our approach, we briefly introduce the two domains in which we operate: large-scale open-source software development and blockchain-based decentralized finance.

Open-source software development. Most large software projects today are developed in the open, with source code hosted on platforms such as GitHub and version-controlled with systems like Git. This development model generates a rich trail of metadata beyond

the code itself. Every commit records what changed, when, and by whom. Over time, a project’s repository accumulates detailed histories of code churn, contributor activity, patch frequency, and the structural evolution of its components [126, 154]. For large foundational projects like the Linux kernel, this history contains decades of commits from thousands of contributors. The metadata is publicly available and machine-readable, yet it is rarely incorporated into security analysis in a systematic way.

Smart contracts and DeFi. The Ethereum blockchain supports executing programs known as *smart contracts*: bytecode programs that run on the Ethereum Virtual Machine (EVM) and whose execution is verified by the entire network [183]. Smart contracts are the building blocks of Decentralized Finance (DeFi), an ecosystem of protocols that provide financial services such as lending, borrowing, and trading of crypto assets without centralized intermediaries [135]. A single DeFi protocol may comprise dozens of interacting contracts, and the ecosystem as a whole manages tens of billions of dollars in user funds [37].

Two properties of this setting are particularly relevant to this dissertation. First, every transaction on the blockchain is permanently recorded and publicly accessible. This means that the complete execution history of any smart contract, every function call, every argument, every storage mutation, is available for analysis. Second, although smart contracts are nominally immutable once deployed, many protocols use *proxy patterns* to achieve upgradeability: a thin proxy contract delegates execution to an underlying implementation

contract, and the implementation can be swapped by the protocol’s administrators. This mechanism is widely used in practice, but it means that audits performed on the original deployment may no longer reflect the code that is actually running.

1.2 Contributions and Dissertation Outline

We identify three classes of retrospective artifacts, each tied to a different stage of the software lifecycle. For each one, we design, implement, and evaluate a system against real-world software and threats. This dissertation makes three contributions, each presented in its own chapter.

Development process metadata (Chapter 2). Every long-lived software project accumulates a detailed record of how its code came to be: commits, patches, contributor histories, code churn, and the temporal patterns of modifications across components. This record encodes signals about code quality that the source alone does not reveal. Functions that have been repeatedly patched tend to harbor further bugs; code touched by many contributors in a short window is more likely to contain inconsistencies; large commits that span many subsystems correlate with elevated complexity and risk [151, 154]. Prior work has studied repository mining for defect prediction [111, 126], but existing approaches have not combined repository-level metadata with code-level metrics at function granularity, nor demonstrated that the resulting predictions can guide automated vulnerability discov-

ery. We present BRAN, a system that learns *bug symptoms*: patterns in both the code and its development process that correlate with the presence of security flaws. BRAN produces a ranked list of functions for targeted auditing or fuzzing. Evaluated on four major codebases (Linux, OpenSSL, FFmpeg, Wireshark), BRAN consistently outperforms prior vulnerability prediction tools. When used to guide the syzkallerkernel fuzzer, it surfaces 26 bugs in the Linux kernel, including 13 previously unknown zero-days.

Execution history from real-world usage (Chapter 3). Once software is deployed, the way it is actually used constitutes a second class of artifact. For smart contracts on a public blockchain, this record is uniquely comprehensive: every transaction, every function call, every storage mutation is permanently recorded and publicly accessible. A protocol’s transaction history reveals which functions are called together under normal conditions, what argument values are typical, and what relationships between inputs and persistent state hold across benign operation. None of this is documented anywhere, and none of it can be recovered from the contract’s bytecode alone, yet it constitutes the de facto definition of how the protocol should behave when it is not under attack. Anomaly detection over execution traces is an established technique [55, 124], but existing DeFi monitoring systems depend on brittle heuristics such as known attacker addresses or fund provenance, or use opaque deep-learning models that lack explainability. We present HOUSTON, an anomaly detection system that extracts semantically meaningful behavioral signals from a

protocol’s own transaction history. HOUSTON comprises two complementary models: one capturing patterns of function invocations that perform sensitive operations, and one inferring likely invariants between function inputs and persistent contract state. Both models are built and continuously refined from live transaction streams. Across 115 DeFi incidents, HOUSTON achieves a 94.8% true-positive rate at a 0.16% false-positive rate, runs within the Ethereum block production window on commodity hardware, and detects attacks without relying on any pre-specified vulnerability patterns.

Version lineage across software updates (Chapter 4). Software changes after deployment, and security-relevant assumptions can shift quietly across updates. In the blockchain setting, contracts that are nominally immutable are routinely upgraded through proxy patterns, and security audits typically cover only the original deployment. Each new implementation inherits a predecessor that, by virtue of having operated without incident, serves as a working reference for acceptable behavior. Comparing a new version against its predecessor transforms single-version analysis into a differential one: rather than asking whether the new code is correct in absolute terms, we ask whether it has relaxed access controls, weakened validation checks, or introduced control-flow and storage-access patterns that the previous version did not exhibit. This formulation catches the class of vulnerabilities that audits and pattern detectors routinely miss, precisely because the individual changes may not, in isolation, be recognizable as bugs. Differential program

analysis has been explored for regression testing and equivalence checking, but no existing tool targets the gap between an audited smart contract deployment and its subsequent upgrades. We present DISSONANCE, a tool that uses symbolic execution to compare successive contract versions and surface behavioral divergences, such as relaxed constraints, broken invariants, and anomalous state writes, that may indicate malicious or unintended modifications.

At the thesis level, these three systems collectively demonstrate that at every stage of the software lifecycle, the system’s own historical artifacts yield a practical and informative behavioral reference for security analysis. The overarching argument is that retrospective analysis is not an ad hoc technique but a coherent and broadly applicable security methodology.

Chapter 5 concludes the dissertation and discusses directions for future work.

Chapter 2

Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms

2.1 Introduction

Software is continually growing in both size and complexity. However, along with new products and features, such growth often comes with new security vulnerabilities, which, when abused by attackers, can drastically affect people's lives. For instance, security experts discovered a high severity vulnerability in Apache Struts, an open-source web development framework used by an estimated 65 percent of Fortune 100 companies, only a year after the same framework had been affected by another vulnerability that had led to a data breach at Equifax, which affected 147 million Americans [84].

Automated Vulnerability Discovery. Security researchers constantly propose new approaches to detect security vulnerabilities in source code before they are discovered and exploited by malicious attackers. Nevertheless, discovering security vulnerabilities takes time, given the size and complexity of modern software. Furthermore, once a vulnerability is discovered, more time is needed for the vendors to develop and release a patch, and for the patch to be deployed on the end systems. During this whole time the software is exposed to attacks. Thus, it is critical to promptly identify the vulnerable portion of large code bases.

However, most of the existing source code analysis tools are insufficient when directly applied to large code bases. In fact, static analysis approaches often need to make a trade-off between soundness and completeness when dealing with complex analysis targets [98, 109]. As a consequence, these approaches either produce too many false positives, making them impractical, or are not generic enough, preventing them from detecting more classes of bugs [102].

To address the aforementioned problems, researchers proposed techniques to guide vulnerability detection tools toward code areas that are likely to be vulnerable, reducing the amount of code to be analyzed. Previous work tried to identify potentially vulnerable code by leveraging software complexity metrics [111], or metrics related to software evolution and maintenance [126, 151]. Others tried to look at syntactic or semantic char-

acteristics of source code that relate to security vulnerabilities [93, 116]. Recently, with the advancement of deep learning techniques, researchers have proposed several methods to automatically learn and extract bug related features from the code [41, 70, 143]. Most of these methods focus solely on the code itself to come up with suspicious code sections, ignoring the crucial fact that any large, long-running software project is the collective effort of a large number of loosely organized engineers over a nontrivial time span. A new comer is more likely to make mistakes than a core developer of a project; a code snippet being modified by a large group of engineers back and forth frequently may incur inconsistency; a large commit spanning many components of a project often indicates high complexity, which may lead to a bug. In another word, the *human factors* buried in repository metadata can server as a great indicator for software quality. Luckily, most (if not all) open source projects use version control systems (VCS) like Git and have their code repositories publicly available. This leaves us rich repository metadata for analysis. By observing the whole history of the studied repository, in part and as a whole, we are able to capture key factors that influence the code quality of the project, based on which we will be able to predict parts of code that are more likely to contain bugs.

Our Approach. We propose a novel approach to characterize potentially vulnerable code by examining *bug symptoms*, which is a collection of both code metrics directly observable from the code and the repository metadata that indirectly reflects code quality. Different

Chapter 2. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms

from previous work on vulnerability detection, which models and identifies the characteristics of vulnerabilities, our approach focuses on the conditions in which vulnerabilities live, and on the different types of symptoms that lead to their occurrence. Our key observation is that vulnerabilities arise from errors that are more likely to occur when either the code or the software development process presents certain patterns.

To capture these patterns, we devise four categories of features to describe functions in large code bases. The four categories focus on code complexity, use of security or quality related primitives, code commit history, and developer characteristics, respectively. Using these features, we are able to train a machine learning model that highlights bug-prone code sections effectively.

We implemented our approach in BRAN, a system that is able to learn different types of bug symptoms for large code bases, and that effectively predicts potentially vulnerable functions. Given a code base, its history, and the list of past, known vulnerabilities (i.e., CVEs), BRAN leverages machine learning techniques to generalize the symptoms that characterize previously discovered bugs, providing a ranking of the functions that are more at risk for undiscovered vulnerabilities. In another word, BRAN looks at the past and the present of a code base to predict its future.

BRAN is not meant to identify the root cause of vulnerabilities, nor is it focused on specific classes of vulnerabilities. Instead, BRAN is designed to assist security experts

and automatic vulnerability detection tools in complex vulnerability auditing tasks by prioritizing high-risk code. In this way, it reduces the search space, making vulnerability discovery more practical.

We evaluated BRAN on four real-world, large code bases (Linux, Wireshark, FFmpeg, and OpenSSL) showing that it is able to predict which functions are likely to be vulnerable. Finally, we used BRAN as a pre-step for fuzzing a recent version of the Linux kernel. We provided syzkaller [39] with a list of 43,473 functions that BRAN reported as likely to be vulnerable, and discovered 26 bugs, out of which 13 have been independently discovered by Google syzbot [60], and 13 are zero days, which would have not been discovered without BRAN’s guidance.

Contributions. In summary, we make the following contributions:

- We propose an approach that characterizes potentially vulnerable code by modeling *bug symptoms*. Our key observation is that vulnerabilities reside in bug symptoms. These symptoms are identifiable when one inspects software from the perspectives of both the code itself and its development procedure.
- Leveraging our approach, we design and develop a system, BRAN, that is able to analyze and predict potentially vulnerable code in large code bases.
- We evaluate BRAN on four large code bases, the Linux Kernel, OpenSSL, FFmpeg, and Wireshark. Results show that BRAN can successfully predict unseen vulnerable

functions. Our evaluation highlights potentially vulnerable functions in a recent version of the Linux Kernel, among which we identify 26 bugs (13 zero days).

2.2 Approach

To highlight the sections of code that are more likely to contain vulnerabilities, BRAN takes two inputs: a database of CVEs, and an open-source code base managed with a version control system. BRAN ingests the input code base and extracts two sets of functions: the set V of vulnerable functions, identified using the CVE database, and the set C of all the functions in the current snapshot of the input code base. By examining CVE records, BRAN builds a ground truth dataset of vulnerable functions (Section 2.3.1). However, we are not sure about whether any function in C is vulnerable or not. A function in C could be either safe or vulnerable. Fortunately, for any real-world code base, non-vulnerable functions outnumber vulnerable functions in C . This is especially true in the case of large, widely used code bases with a long history. In light of this observation, BRAN can assume that a randomly sampled set of functions $NV \subset C$ is non-vulnerable.

BRAN extracts code metric and repository metadata features from both function sets. Using V as positive examples and NV as negative examples, BRAN trains a regression model that takes in a function feature vector and generates a vulnerability score. During vulnerability search, users may first use BRAN to highlight the parts of code that are more

likely to be buggy. Then, they may employ more computationally expensive tools such as fuzzers targeting only the highlighted code sections for more fine-grained vulnerability information. Users are able to tune the threshold of the vulnerability score based on their computational budget. With a suitable threshold, users get the chance to capture the majority of vulnerabilities while testing only a small fraction of the repository.

2.2.1 Code Metric Features

Intuitively, large, complex functions are more likely to have unforeseen corner cases; code that shows known bad practices (e.g., no sanity checks on input parameters, many casting operations) is more prone to bugs such as memory corruption issues. BRAN quantifies these intuitions using *code metric features* as summarized in Table 2.1 under category *Complexity* and category *Function Properties*.

Complexity. Complex functions, i.e., functions performing many operations through non-trivial control flows, are difficult to debug and maintain, and thus are more likely to contain bugs. To some extent, the complexity of a function is also related to the complexity of the module it belongs to. Functions that live in complex modules have a greater chance of encountering errors, giving rise to vulnerabilities [25, 112, 151, 152].

Function Properties. Certain function characteristics might indicate a higher probability of the presence of bugs. Functions may contain examples of bad coding practices (e.g.,

many casts (**F6**), lack of comments (**F9**) [8]). Or they may present a large attack surface (e.g., long parameter list (**F7**), lack of sanity checks (**F10**)).

2.2.2 Repository Metadata Features

While code metric features describe quality and security related properties of the code directly, we can also leverage the rich repository metadata information to derive useful signals for vulnerability prediction. We devise the set of repository metadata features as summarized in Table 2.1 under category *History* and category *Developers & Reputation*.

History. The history of a function also provides hints about the likelihood of the function containing bugs. A function or module that is frequently patched usually plays a critical role in the project. Frequent modification also makes code more prone to flaws (**F11**, **F12**) [114, 151]. In version control systems such as Git, a commit is the basic unit for content update. Commits with radical changes (**F13**, **F14**) usually imply major code refactors, which may introduce new bugs or violate established conventions in the code. Commits that involve too many parts of different components (**F13**, **F14**) indicate a tightly coupled design in the project. With more moving parts, tightly coupled components are usually more bug-prone than independent ones.

We note here that bugs related to a hard-to-maintain commit are not necessarily introduced by this commit itself. It could be the commit before or after this commit that

actually implants the bug. Therefore, predicting vulnerabilities using single commits as the basic unit [126] is not enough. We try to capsule bug symptoms like this by using functions as BRAN’s basic operation unit. A function carries consistent semantics throughout its lifetime in the development procedure. More importantly, most of the time, it honestly reflects the design changes in the code. By aggregating the history of a function, BRAN is able to capture the correlation among multiple commits and program components.

Developers & Reputation. Traits of a developer may reflect the quality of their code. For instance, well-known or experienced developers generally write more reliable code. The trustworthiness of a developer can be estimated through various indicators, such as their average number of contributions (**F21**, **F22**), and their popularity in the community (**F19**). Sometimes, collaboration among developers may also come into play. For example, having a larger number of developers contributing to the same function is against the separation of concerns and responsibilities principle, which, in turn, is an evidence of poor project management (**F17**, **F18**). In addition to the programmer information fetched from Github, we also estimate developers’ expertise in terms of their contributions towards the repository being studied (**F24**) as some experienced engineers do not keep an active public profile. Feature (**F24**) also reflects the level of knowledge (familiarity) a developer has for the project.

Table 2.1: Features computed by BRAN grouped by category.

Feature	Description & Rationale
Complexity	
(F1) Function lines of code (LOC)	Long functions are more difficult to debug and maintain, thus are more prone to bugs.
(F2) Cyclomatic complexity	Complex functions are more prone to vulnerabilities. We measure the McCabe’s complexity [105].
(F3) Module LOC	LOC of the module that the function belongs to. Using large modules is not a good practice.
(F4) Module complexity	Sum of the McCabe’s cyclomatic complexity of all the functions in the function’s module.
(F5) N. co-located functions	Number of functions in the function’s module. Using large modules is not a good practice.
Function Properties	
(F6) N. casts	Casting is known to be a source of bugs.
(F7) N. input parameters	More function input parameters imply a larger attack surface.

Continued on next page

Chapter 2. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms

Table 2.1 continued from previous page

Feature	Description & Rationale
(F8) N. local variables	Declaring many local variables is a sign that the function performs many tasks. Complex functions are more prone to vulnerabilities.
(F9) N. lines of comment	Lack of comments is a bad practice.
(F10) N. input sanity checks	Sanity checks on input parameters reduce the function's attack surface.
History	
(F11) N. past changes	Frequently patched functions may be poorly written and maintained or play critical roles.
(F12) N. module's past changes	Frequently modified modules may contain critical tasks or be poorly written and maintained, leaving functions in a more error-prone environment.
(F13) N. second largest commit addition	Commits with more lines of addition usually indicate more radical change in the code. As the commit with the largest number of addition is mostly the first commit, we take the second largest commit addition number.

Continued on next page

Chapter 2. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms

Table 2.1 continued from previous page

Feature	Description & Rationale
(F14) N. largest commit deletion	Commits with more lines of deletion usually indicate more radical change in the code.
(F15) N. max commit module span	A commit that spans across more modules tend to be more complex and therefore more bug-prone.
(F16) N. max commit function span	A commit that touches more functions tend to be more complex and therefore more bug-prone.
Developers & Reputation	
(F17) N. function's contributors	A reason for having a modular software is separation of concerns <i>and</i> of responsibilities. Too many developers contributing to the same function may cause issues.
(F18) N. module's contributors	Too many developers contributing to a module may indicate bad team management.
(F19) Avg. contributor followers	Well-known developers in the open-source community are generally more reliable. Open-source hosting services like GitHub allow developers to <i>follow</i> other developers.

Continued on next page

Table 2.1 continued from previous page

Feature	Description & Rationale
(F20) Avg. contributor stars	Open-source developers can <i>star</i> projects in public repositories (e.g., GitHub) to express appreciation. We consider the average number of stars received by all the contributors of a given function. Each contributor's stars are counted as the sum of the stars of all the public repositories they own.
(F21) Avg. contributor forks	We consider the average number of forks of all the contributors of a given function, counted as the sum of the forks of all public repositories that they own.
(F22) Avg. contributor public repositories	Developers who frequently contribute to the open-source community are generally more experienced. We consider the average number of public contributions across all of the function's contributors.
(F23) Avg. contributor watchers	Watching a repository means receiving notifications about activities of the repository. The number of watchers reflects the popularity of a project, which can be an estimation of the trustworthiness of its developers. Here, we consider the average number of watchers of all of the function's contributors.

Continued on next page

Table 2.1 continued from previous page

Feature	Description & Rationale
(F24) Min. contributor’s contribution	The number of contributions within the project of the least experienced developer among all developers contributing to this function. Experience is measured by the number of commits a developer has made for the project.

2.2.3 Dataset

The dataset quality is essential to the performance of data-driven vulnerability prediction techniques. With inappropriate datasets, even advanced machine learning algorithms will render ineffective. For instance, small datasets may provide too little information for models to learn proper features automatically, and incorrectly labeled datasets give wrong feedback to models while training. All these defects in datasets may lead to poor performance on real code bases.

Past work on vulnerability-type agnostic, machine learning-based vulnerability prediction techniques relied on flawed datasets. Among them, Chernis and Verma [24] used only 100 vulnerable functions during training; VulSniper [41] used only two types of vulnerabilities in the SARD project [11]; Harer et al. [70] and Russell et al. [143] collected dataset from sources including GitHub [57] public repositories and Debian Linux distribu-

tion [36]. The size of their dataset is approximately 12 million functions. However, they labeled the dataset with static analyzers like Cppcheck [2] and Flawfinder [4], which are known to have high false positive or high false negative rates (i.e., incorrect labeling).

In this chapter, we make use of a dataset that comprises four large, real-world projects: the Linux kernel, OpenSSL, FFmpeg, and Wireshark. For each project, we collected the current code repository, its history, and its corresponding CVE reports. We labeled vulnerable functions based on CVE reports, and trained BRAN to predict future vulnerabilities based on history knowledge. In this way, our dataset is able to provide BRAN with an accurate and complete knowledge about vulnerabilities.

2.3 Implementation

In this Section we discuss relevant aspects of BRAN’s implementation. We first outline the process used to extract vulnerable functions out of a code base, and then detail the implementation of BRAN’s machine learning model.

2.3.1 Inputs Processing

Unlike other automated vulnerability detection systems (e.g., a static analyzer), which may introduce a large amount of false positives into the dataset, the CVE database is a reliable source for vulnerability-related information. Therefore, as mentioned in Section 2.2,

Chapter 2. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms

BRAN relies on the CVE reports of the input code bases to create a ground truth dataset of vulnerable functions. In particular, BRAN is able to enact an automated ground truth generation process for any software product P (e.g., OpenSSL) that: 1) P is open-source; 2) P uses git for version control; 3) P is hosted on GitHub. Note that the same data preparation logistics can be adapted to other version control systems and hosting services as well. To extract vulnerable functions, BRAN performs the following nine steps:

1. Fetch a local copy of P 's code base with `git clone`.
2. Download lists of CVEs associated with P from the CVE public database [123].
3. For each CVE associated with P , refer to the National Vulnerability Database (NVD) for the git patch that fixed the vulnerability. The patch information (SHA-1 hash of a git commit) is not always available, as there might be missing data or simply the CVE might not have a patch yet.
4. If a given CVE has an associated fixing commit c , run `git checkout c` to retrieve the corresponding revision.
5. Extract all files modified by c and rename the patched files. For instance, a file with name `sample_file.c` gets copied to `sample_file.fixed.c`.
6. Run `git checkout c^` to retrieve the revision right before c was applied.

7. Extract and copy the non-patched version of all the files modified by `c` (e.g., `sample_file.c` is copied to `sample_file_unfixed.c`).
8. For each (`*_unfixed.c`, `*_fixed.c`) pair, identify functions that are modified by `c`.
9. Extract the body of each function identified from the corresponding unfixed file, i.e., the version of the function right before `c` was applied.

Following the steps above, we generate a dataset of vulnerable functions. Furthermore, we can configure the above procedure to extract vulnerable code considering only CVEs within a given time span (for example from 2012 to 2014) or considering only specific submodules of the code base.

It is worth mentioning that VUDDY [81] proposed a similar vulnerable code extraction process. The main difference between the two approaches lies in the way they identify commits associated to CVEs. While we rely on the National Vulnerability Database (NVD), allowing BRAN to retrieve patches for CVEs precisely, VUDDY looks for commit messages containing the “CVE-20” string. It then filters out irrelevant commits (i.e., commits that contain “CVE-20” but did not fix any CVE) using a number of heuristics (e.g., excluding merging commits and reverting commits). VUDDY’s heuristics-based approach may cause the resulting dataset to be inaccurate and incomplete. Indeed, irrelevant commits may be included in the dataset while relevant commits may be ignored.

In addition to the NVD, BRAN allows for the integration of other sources that provide a mapping between CVEs and git commits. For example, in the case of the Linux kernel, we augment the NVD with the mapping provided by the `nluedtke/linux_kernel_cves` GitHub repository [119], which allows us to retrieve 168 more commits than those retrievable by solely relying on the NVD.

2.3.2 Learning Bug Symptoms

We compute code metric features and repository metadata features with a set of external tools and services. More specifically, we use Joern [187] to extract features **(F5)**, **(F6)**, **(F7)**, **(F8)** and **(F10)**. We feed Joern with the source code of all the functions for which context properties need to be extracted. Joern treats the set of functions as an entire program and produces the Program Dependency Graph (PDG). We then query the PDG to extract context properties. Through multiple `git` operations, we compute features **(F11)**, **(F12)**, **(F17)** and **(F18)**. Features **(F13-F16)** and feature **(F24)** are extracted with the help of `git log`. Features **(F19)**, **(F20)**, **(F21)**, **(F22)** and **(F23)** are computed by querying code bases through GitHub’s public APIs [58]. These features are fairly straightforward counts of events or basic properties. Features **(F1)**, **(F3)** and **(F9)** are collected directly from raw code files. Features **(F2)** and **(F4)** are computed with `pmccabe` [132], which provides a complexity score for functions and modules.

Code base context properties extraction is implemented as a parallel and distributed data pipeline on top of Apache Flink [1]. This allows us to efficiently process large code bases. We use random forest as the classification model for context properties representation, as this gives us a score which lies between 0 and 1, rather than a binary output. Although we also experimented with others classifiers (e.g., a neural network composed of a sequence of dense layers), we did not experience significant performance improvements. Using scikit-learn, BRAN trains a random forest classifier over the vector representation of input functions. In particular, BRAN uses the extremely randomized trees training algorithm. The number of trained tree is 1000.

2.4 Evaluation

In this section, we evaluate BRAN and demonstrate that, by studying both code metric and repository metadata features, BRAN is able to effectively reduce the search space of vulnerability search. We organize this section around the following research questions:

RQ1: *Is BRAN effective in predicting previously unseen vulnerabilities?* We evaluated the vulnerability prediction ability of BRAN comparing to previous work. We considered four large real-world code bases: the Linux kernel, OpenSSL, FFmpeg, and Wireshark. Results indicate that BRAN is effective in identifying previously unseen (i.e., not in the training set) vulnerabilities (Section 2.4.2). We compared our system to existing tools,

demonstrating that it outperforms state-of-the-art approaches (Section 2.4.3).

RQ2: *How does the non-vulnerable to vulnerable functions ratio in the training set affect machine learning-based methods?* The severe imbalance between vulnerable and non-vulnerable functions poses a practical challenge for machine learning-based vulnerability detection methods [40]. In search of a proper ratio for training set, We evaluated BRAN on different non-vulnerable function to vulnerable function ratios. Results suggest that a severe imbalance in the training set does indeed harm performance. However, we show that effective machine learning models can still be trained considering only part of the data, overcoming the imbalance issue (Section 2.4.4).

RQ3: *Are learned bug symptoms unique to the training code base? Otherwise, are they generalizable to other code bases?* Although the choice of feature set is generically applicable to different code bases, whether the trained model on one code repository is effective for another repository is not clear. We evaluated models trained using one code repository and applied them to a different repository. Results show that bug symptom knowledge is generally not repository agnostic. (Section 2.4.5).

RQ4: *Is BRAN effective in assisting large scale software testing?* We show one of the many possible applications of BRAN in the context of guided fuzzing (Section 2.4.6). We configured a fuzzer to prioritize functions marked as highly-vulnerable by BRAN. In this

way, BRAN restricts the exploration space of the fuzzer. Experimental results show that the fuzzer guided by BRAN significantly outperforms the unguided fuzzer.

2.4.1 Datasets

We evaluated BRAN against four large, real-world projects: the Linux kernel, OpenSSL, FFmpeg, and Wireshark. For each of these projects, we considered the current code repository, its history, and the corresponding CVE reports.

To evaluate the performance of BRAN, we have the following setup. Given the snapshot of a code base at time T in its history, we collect a *training set* for BRAN for model training, and an *evaluation set* for performance evaluation. The *training set* is composed of the vulnerable functions that appeared in CVE reports before time T (as vulnerable examples), and a set of functions sampled from the code base snapshot at time T (as non-vulnerable examples). The data collection procedure is described at the beginning of Section 2.2. We tried to keep a balance between the vulnerable and non-vulnerable classes in the training set by controlling how many non-vulnerable functions we sample (Section 2.4.4). The *evaluation set* is used to evaluate the performance of machine learning models in finding vulnerabilities that were not known at time T . More specifically, to create the ground truth dataset of vulnerable functions, we look at the CVEs that have been reported *after* time T . The primary objective of our evaluation is to study how many

of such functions are correctly predicted by BRAN. The non-vulnerable functions in the evaluation set are all the functions belonging to the code base snapshot at time T , with the exclusion of those functions sampled to build up the training set and the functions that were included in the vulnerable set. Note that not all functions belonging to this set are guaranteed to be non-vulnerable. Some of them might actually be vulnerable, and the vulnerability has not yet been discovered.

To answer our research questions, we gathered different *training sets* and *evaluation sets* under two settings:

Setting I. $T = \text{end of 2015}$: In this setting, we use knowledge up to the end of year 2015 to predict vulnerabilities that were discovered in 2016 or later. By setting T back to 2015, we were able to accumulate enough CVEs to build a vulnerability ground truth for the evaluation set. We answer questions **RQ1-RQ4** with experiments under this setting.

Setting II. $T = \text{end of 2018}$: In this setting, we used knowledge up to the end of year 2018 to predict vulnerabilities that were not known at that point in time. In this setting, we do not have a sufficiently large ground truth of vulnerable functions in the evaluation set, as only few vulnerabilities have been reported since the end of 2018. Therefore, the only way to confirm BRAN’s predictions is to check them with testing tools ourselves. We use this setting to answer question **RQ5**.

Table 2.2: Number of functions in each dataset.

T	Set	Class	Dataset			
			Linux	OpenSSL	FFmpeg	Wireshark
2015	Training	Vuln	1,473	91	167	127
		Non-Vuln	8,838	546	1,002	762
	Evaluation	Vuln	927	128	107	388
		Non-Vuln	257,290	4,965	14,081	24,419
2018	Training	Vuln	2,400	219	274	515
		Non-Vuln	14,400	1,314	1,644	3,090
	Evaluation	Both	271,624	6,243	16,377	25,367

The size of each dataset generated in the two settings is reported in Table 2.2. For each code base studied, we balanced the non-vulnerable to vulnerable function ratio to 6 for the training set (including validation) (see Section 2.4.4). In this chapter, we use the output of the classifier as a vulnerability score for the input function. Higher scores indicate a better chance to find a vulnerability in the input function.

2.4.2 Prediction Capabilities

In this section, we use **Setting I** to answer **RQ1** about BRAN’s ability to predict vulnerable functions. Under this setting, we are interested in how many of the functions that

Chapter 2. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms

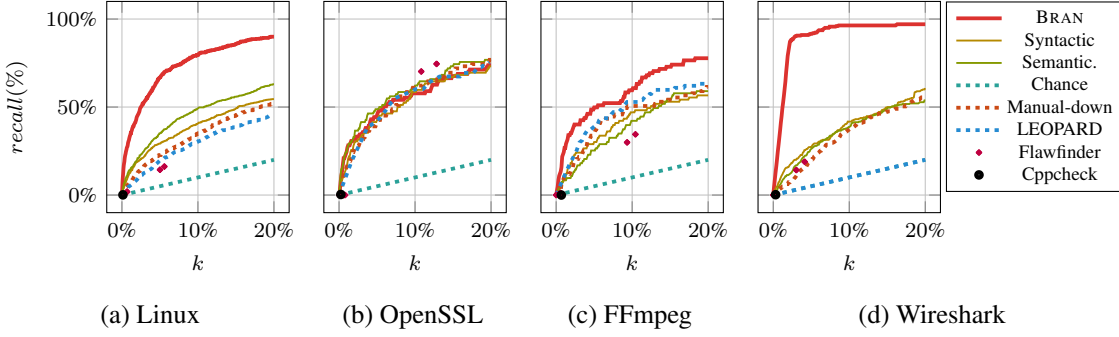


Figure 2.1: Models trained using data from a single code base and then used to rank functions in the same code base: recall on predicting vulnerable functions over top $k\%$ of ranked functions. $k \in [0, 20]$.

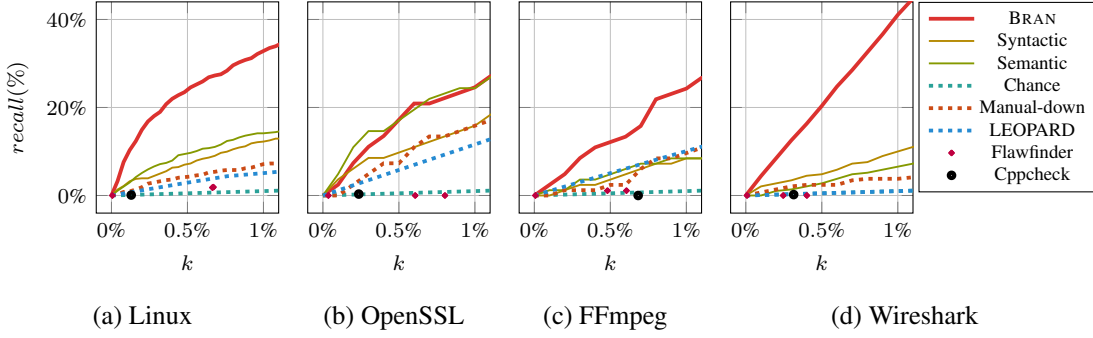


Figure 2.2: Models trained using data from a single code base and then used to rank functions in the same code base: recall on predicting vulnerable functions over top $k\%$ of ranked functions. $k \in [0, 1]$.

are found to be vulnerable *after* 2015 are correctly predicted by models trained with the knowledge available *up to* 2015. Formally, let VP be the set of functions that are predicted as vulnerable by BRAN and LV be the set of vulnerable functions in the evaluation set; we calculate the *recall* of BRAN on the set LV as $recall = |VP \cap LV|/|LV|$ as the primary metric for effectiveness. Note that, since some bugs in the code base remain unidentified, *recall* in this experiment is an underestimation of BRAN’s performance.

A key aspect of computing *recall* is how we define VP , i.e., based on what criteria BRAN judges a function to be vulnerable. BRAN assigns a vulnerability score (a probability of being vulnerable) to each function. Therefore, naturally, we can set a vulnerability score threshold γ for each code base, and label functions with a score higher than γ as vulnerable. How to pick a suitable γ , however, is not trivial. In the scenario of large-scale automated vulnerability discovery, the number of functions that one is able to inspect depends on the available computing resources. Therefore, the objective of BRAN is to maximize *recall* given the maximum size of VP as imposed by the available computing budget. We evaluate *recall* for different values of γ to simulate various computing budgets. More specifically, we ranked all functions according to their vulnerability score and considered the higher-most $k\%$ as set VP . We set k to $[0, 20]$ and $[0, 1]$ respectively and show the results of all sub-models together with BRAN combined model in Figure 2.1 and Figure 2.2.

As shown in Figure 2.1, with the help of BRAN, by inspecting only the top 20% functions in a code base, security experts are able to cover more than 70% of the vulnerabilities in all the four considered code bases. For more time-consuming human inspections, users can conveniently set the threshold of BRAN to a higher value and only invest in high-confidence suspicious functions. As Figure 2.2 shows, by inspecting only 1% of the code base, BRAN achieved more than 20% recall of vulnerabilities on all four code bases tested.

We conclude that BRAN is able to effectively distinguish vulnerable functions from non-vulnerable ones.

Precision Analysis. BRAN is not designed to pinpoint bugs as a stand alone tool. Instead, it assists other vulnerability discovery tools in reducing their search space (as we will demonstrate in Section 2.4.6). While classification problem is generally evaluated with both recall and precision, avoiding a high false positive rate in our case is particularly challenging but not rewarding. We therefore choose not use precision as a key metric of the effectiveness of BRAN.

2.4.3 Comparison with Existing Tools

To further assess the performance of BRAN, we compared it against existing tools for source code vulnerability prediction.

Code metric based approaches. We compared BRAN against two code metric baselines, namely the manual-down approach and LEOPARD [40]. The manual-down approach ranks functions by their source lines of code in descending order and labels longer functions to be more prone to bugs. Simple as it is, past work has shown that manual-down approach achieves similar or even better results compared to state-of-art vulnerability prediction models [40, 83, 206]. LEOPARD is a recent tool that leverages both complexity and vulnerability metrics in code defects detection. It first categorizes target functions into

a set of bins based on the complexity metrics, then ranks functions in each bin by their vulnerability metrics (e.g. number of pointer operations), and labels the top n functions in each bin as vulnerable.

We implemented manual-down approach ourselves and the authors of LEOPARD kindly tested LEOPARD on our dataset. Manual-down processed all functions successfully; LEOPARD failed to process the Wireshark dataset. The results of both baselines are shown in Figure 2.1 and Figure 2.2. In Figure 2.1 where the top $k\%$ ($k \in (0, 20]$) functions were considered vulnerable, BRAN’s outperformed metrics baselines in Linux, FFmpeg, and Wireshark dataset and got comparable results in OpenSSL dataset. In the high-confidence zone where only the top 1% functions were inspected (Figure 2.2), BRAN achieved twice the recall of any of the tools on three of the four code bases. Similar to the baseline techniques, BRAN also used code metric features for classification. However, in addition to code metrics employed by both manual-down and LEOPARD, BRAN further takes the repository metadata features into consideration. With richer context features, BRAN is able to predict vulnerabilities more effectively.

Static analyzers. We ran two popular static source code analyzers, namely Flawfinder [4] and Cppcheck [2], on our **Setting I** dataset as baseline comparison. Both Flawfinder and Cppcheck examine C/C++ source code for possible bugs and undefined behaviors by matching a built-in database of known problems. Nevertheless, BRAN and such static

analyzers are designed differently. BRAN assigns a continuous score to each analyzed function, effectively producing a ranking. It is then up to the users to set a threshold score according to their time and computational budgets to further analyze only top $k\%$ of the functions. On the other hand, static analyzers make binary decision for matches, such that a function is either suspicious (i.e., leads to warnings or errors being raised) or safe. This still leads to only $k\%$ of the input functions being further analyzed by security experts, with k being the ratio between the number of reported suspicious functions and the total number of analyzed functions. In both the cases, given $k\%$ of the input functions, we use recall, i.e. how many of these suspicious functions are actually vulnerable, as the performance measure. For Cppcheck, we only have one single value for k , which reflects the number of reported suspicious functions. Flawfinder allows users to choose from 5 *risk levels*, which in turn gives 5 different values of k in the test.

Results for Cppcheck and Flawfinder are shown in Figure 2.1 and Figure 2.2. BRAN outperformed both baselines by a large margin in all code bases except OpenSSL dataset. For low budget inspection assistance, where only the top 1% functions are considered interesting, both analyzers gave near-zero recall while BRAN was able to provide a recall of more than 30%. One interesting observation is that Cppcheck exhibited only near-zero recall for all code bases. This might be due to the fact that while Cppcheck strives to achieve a low false positive rate, it inevitably suffers from low true positive rates [2]. We

note here that we are aware that it seems unfair to compare Flawfinder and Cppcheck to BRAN in terms of recall as they are designed to match vulnerabilities in a much more fine-grained way. However, these are the ones among others in this category that are close to BRAN’s design goal. We believe that the comparison adds interesting perspectives to discussion.

We conclude that current static analyzers only match known patterns and are sub-optimal in the setting of vulnerability prediction compared to BRAN.

Automatic feature learning. With the recent advancement of deep learning, researchers also propose to locate suspicious code sections by studying code properties with neural networks. VulSniper [41] and Russell et al. [70, 143] leveraged neural networks for type-agnostic bug finding, which is similarly to BRAN. We contacted both groups for help in reproducing their results on our dataset but we were turned down unfortunately.

To compare BRAN to automatic feature learning methods, we designed and implemented two neural network models ourselves based on previous work and compared them to BRAN. The first model extracts syntax pattern [93] of functions with tree-shaped convolutional neural network [113] by abstract syntax tree vectorization. We refer to this model as *syntax model* for the remainder of this chapter. The second model captures semantically relevant vectorial representation of functions. More specifically, it leverages the word2vec [110] approach from the natural language processing (NLP) community and

embeds the functions as a paragraph [143]. We refer to this model as *semantic model* for the remainder of this chapter. The word *semantic* here is borrowed from NLP. It does not refer to *program semantics* as defined in the programming language community. Both models are trained on the same dataset as BRAN as binary classifier.

In the syntax model, the convolution layer of the syntactic model deployed 128 kernels (feature detectors). The classification sub-network was a fully connected network with one hidden layer of width 20. The model was trained for 30 epochs with batch size 16. Learning rate was set to 0.001. In the semantics models, the one-dimensional convolution layer deployed 16 kernels, each of the size of 3 words, and used a ReLU activation function. The LSTM layer was initialized using 50 memory units. The final dense layer used a sigmoid activation function. The model was trained for 5 epochs with batch size of 50. In our experiments, different configurations did not show significant performance improvements.

For the high budget scenario shown in Figure 2.1, BRAN achieved similar results in the OpenSSL dataset comparing to both the syntax and the semantic model. For the other three datasets, BRAN greatly outperformed both models. In the low budget case as shown in Figure 2.2, BRAN beat the syntax model in all four datasets. The semantic model displayed comparable result as BRAN on OpenSSL dataset, but only got half the recall on the other three datasets.

As a side note, both neural network based models generally outperformed the two code metric based methods. While automatic feature learning is powerful in extracting useful code patterns, with the help of repository metadata, BRAN achieved better results in the benchmarks.

2.4.4 Training Set Balance

Known vulnerable functions are rare in real-world projects compared to the total number of functions in a code base. This imbalance between vulnerable and non-vulnerable code hinders the adaptation of machine learning based methods for vulnerability detection [40]. The reason is that an imbalance in training data often results in poor classification accuracy.

In practice, fortunately, we do not have to stick to the actual ratio between non-vulnerable function and vulnerable function (denoted as R) in the code base when it comes to training. For BRAN, we tuned R in the training set by sampling only a part of non-vulnerable functions. Empirically, we tested several R values to search for the best R setup for training.

Figure 2.3 shows how data balance in the training set R influences vulnerability detection performance for BRAN. We found that models trained with R between 3 and 6 generally outperform other models by a notable margin. Much higher or too low R values

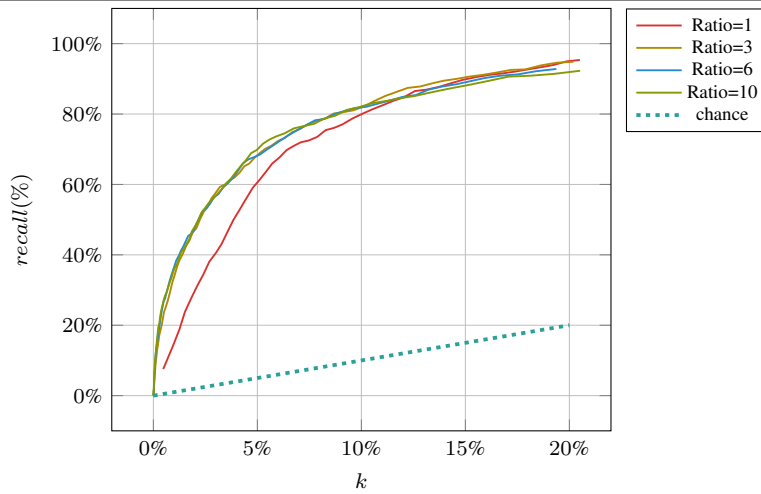


Figure 2.3: Models performance with various non-vulnerable function to vulnerable function ratio in the training set: recall on predicting vulnerable functions among the top k % of ranked functions.

have negative effects on the classification performance. The result also indicates that machine learning based methods are in fact practical in use cases where vulnerabilities are very sparse.

2.4.5 Bug Symptoms Knowledge Transfer

An interesting question to consider when modeling bug symptoms is whether such symptoms are code-base-dependent. If bug symptoms are shared across code bases, it is possible to use models trained on one code base to predict vulnerabilities in other ones. In other words, if bug symptoms are *transferable* across code bases, the resulted machine learning models would be generic, drastically widening their scope of applicability.

To assess how well bug symptoms generalize, we repeated the experiment discussed in Section 2.4.2 (again under **Setting I**). Only this time, instead of using models trained on one code base to predict future vulnerabilities in the same code base, we used models trained on a different code base. For example, we used the model learned from Linux using context properties to predict vulnerable functions on OpenSSL, FFmpeg, and Wireshark. We did this for all 12 transfer pairs (training code base $\rightarrow$ prediction code base). The results are shown in Figure 2.4. Each subplot also reports as a baseline the performance of the *native* model, whose training code base is the same as its prediction code base.

From Figure 2.4 we notice that, for half of the target repositories, native models delivered the best performance, while for the other half of the target repositories, one (and only one) of the transferred models outperformed the native model. Considering all transfer pairs tested, most of the time transferred models had poor performance. Sometimes the transferred model was even worse than chance (e.g., Linux $\rightarrow$ Wireshark). This indicates that code metrics and repository metadata features like traits of developers are highly correlated to the code base being analyzed.

To conclude, bug symptoms learned from one code base have a certain level of ability to be transferred to a different one. However, in most cases, using the symptoms learned from the same repository derives better results.

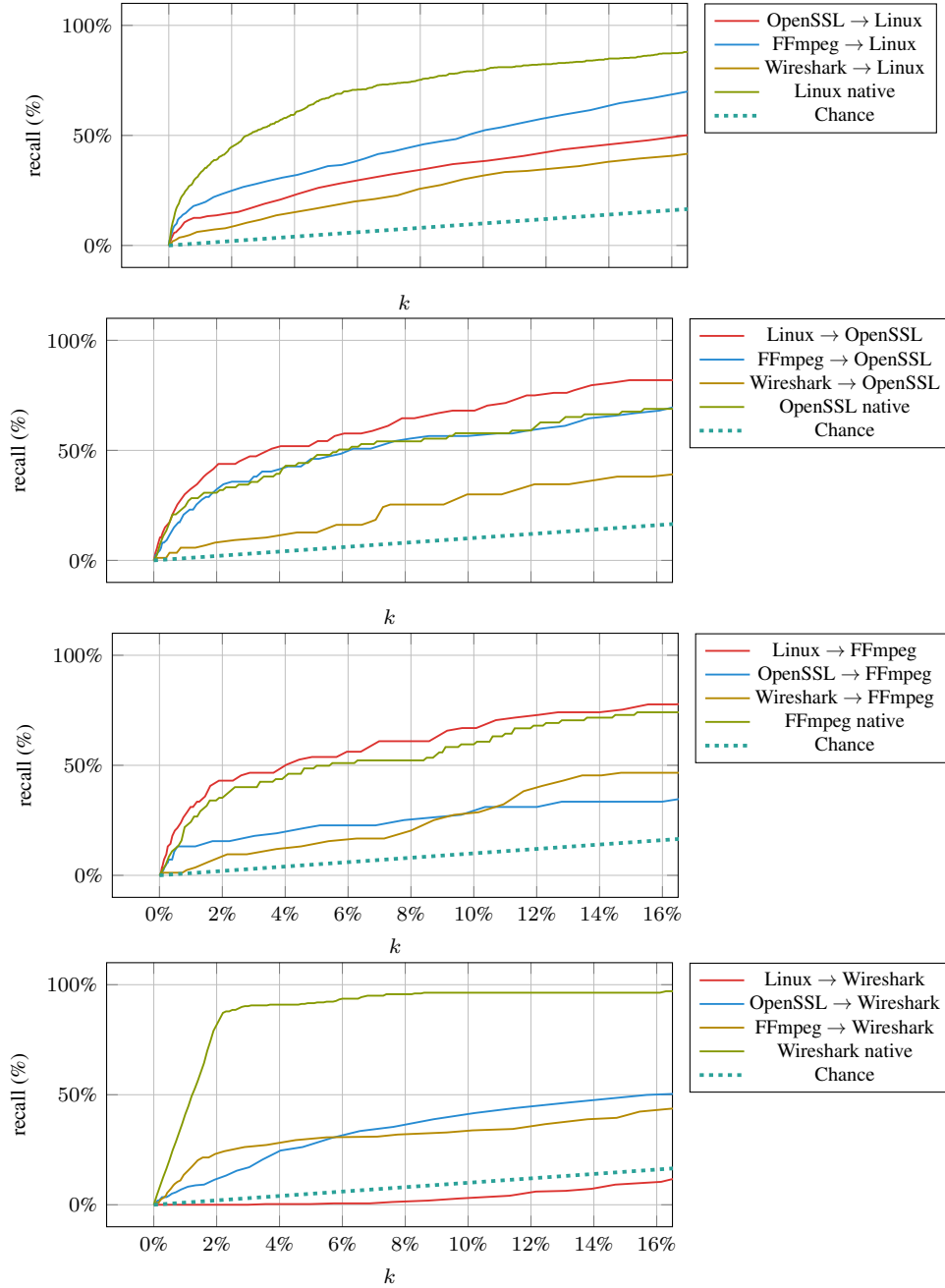


Figure 2.4: Models trained using data from one source code base and used on a different target code base (source → target): recall on predicting vulnerable functions among the top $k\%$ of ranked functions.

2.4.6 BRAN-guided Fuzzing

In this section, we want to understand the effectiveness of BRAN in helping automated testing tools to find new bugs in very large and well-tested code bases. Thus, we present a real-world scenario where BRAN can be directly applied. Specifically, we use syzkaller [39], a widely used Linux kernel fuzzer, in two configurations:

syzkaller: The recommended configuration as suggested in the official documentation [178].

syzkaller_{bp}: syzkaller guided toward fuzzing functions marked with high confidence as vulnerable by BRAN.

To configure *syzkaller_{bp}*, we first obtained all the functions that BRAN marked as vulnerable with more than 80% confidence. These is our initial set of functions (I). We want the fuzzer to generate inputs that have a high likelihood of exercising such functions. An input from user space can reach a function f in I either directly (if f is a `syscall` [9]) or through one of its callers. Therefore, given the call-graph [145] of the Linux kernel and the initial functions (I), we obtained all the additional functions (A) that can reach, in the call-graph, a function in I .

Any input that reaches a function in A has the potential to reach a function in I . Thus, we added the functions in A to the list of the functions used to guide syzkaller. Doing so,

we obtained a list of $T = I \cup A \approx 100K$ functions—compared to the $\approx 500K$ functions in the Linux Kernel. However, the actual number of functions that get compiled in the kernel is much less and depends on the corresponding kernel configuration [197]. This reduces T , the list of functions that need to be fuzzed, to 43,473 functions. Finally, we modified the `kcov` [177] pass in `gcc` [65] to instrument only the functions T , so that `syzkaller` prioritizes the inputs that have a high likelihood of exercising the target functions.

We ran our customized *syzkaller_{bp}* configuration for 48 hours. On the other hand, to reduce any non-deterministic effect and to perform a stronger comparison, we ran the default *syzkaller* configuration for twice as long, i.e., 96 hours. Table 2.3 shows the total number of crashes we found using the two fuzzer configurations on the latest version of the Linux kernel (April 2019). In total, the fuzzer guided by BRAN (*syzkaller_{bp}*) found, in half the time, 26 bugs compared to only 2 bugs found by the default configuration (*syzkaller*). This shows that BRAN is effective in guiding fuzzers to find *more* bugs *quickly*. Indeed, 18 (out of 26) functions where we found crashes were actually predicted by BRAN to be vulnerable with 100% confidence. Among such 26 bugs, we found that 13 of them have been independently discovered by Google syzbot [60], and 13 are previously unknown bugs.

2.5 Discussion and Limitation

Ground truth dataset. Despite our efforts to provide BRAN with appropriate ground truth during the training phase, the collected datasets might still bear some inaccuracies, which might introduce a bias in BRAN’s predictions, and in turn affect its effectiveness. In particular, when extracting vulnerable functions from CVE reports, we label all functions modified in the patching commit as vulnerable. However, in some cases a patch modifies multiple functions, but only some of them are directly related to the actual root of the vulnerability. In future work, we may refine our procedure to precisely identify vulnerable functions by leveraging available sources of information, e.g., CVE descriptions and commit messages [196].

Similarly, to label non-vulnerable functions, we rely on the assumption that, given a snapshot of a large code base, the vast majority of the functions can be assumed to be safe. Thus, a random sample of the functions in a code base can be used as a ground truth for non-vulnerable functions. Inevitably, such a ground truth might contain functions that are indeed vulnerable but are not discovered yet, introducing inaccuracies. Fortunately, for very large, well-maintained code bases, we can expect this to be rare. As a matter of fact, by improving the procedure we use to label our training set, BRAN’s performance would further increase.

Distinguish types of bugs. By design, BRAN is only able to detect potentially vulnerable functions, without providing information about the type of vulnerability. Future work could address such limitation by extending BRAN to distinguish different classes of bugs. However, this would require a labeled training dataset.

Precision and recall. BRAN incurs a relatively high false positive rate compared to stand alone bug finding tools such as Flawfinder and Cppcheck. However, as a search space reduction tool, BRAN is not designed to achieve high precision. When being compared to tools like LEOPARD with similar design objectives, BRAN delivered higher precision as well as higher recall.

Identification of functions. BRAN uses function as the basic unit for repository metadata analysis, and relies on the pair of file path and function name as the unit identifier. This abstraction introduces inaccuracy when the function or the file is refactored to a different name. We believe that the accuracy of the analysis can be further improved by tracking renames and moves. For the purpose of this chapter, however, this simplified abstraction already works well enough.

Cold start problem. BRAN may suffer from the cold start problem where new functionalities in the project may have fewer metadata to base the analysis on compared to the long-running ones, causing BRAN to have a slight bias towards code with longer history. While we admit that BRAN has a insignificant systemic bias favoring older functionali-

ties, we point out that this bias is reasonable as there is simply more chance of making mistakes for functionalities with longer history. Also, as most features BRAN uses (all of the code metric features and many of the repository metadata features (e.g., author reputation)) are in fact unbiased against new functions, BRAN should be able to handle newly added functionalities reasonably in practice.

2.6 Related Work

Pattern-based. Pattern-based approaches make use of already-known vulnerability patterns to find potential vulnerable code. Some of the popular tools are Flawfinder [4], PScan [6], RATS [7] and ITS4 [176]. Though these tools are very efficient and practical, they incur very high false positive and false negative rates, because they often fail to identify complex vulnerability patterns. Since then, several efforts have been made to build more advanced static analysis techniques [68, 89, 99, 159, 175, 186, 188, 190, 192] for pattern-based vulnerability detection. However, these techniques rely on previous knowledge of vulnerability patterns and thereby fails to discover new ones. In addition, there are approaches [106, 164, 191] that aim to find vulnerabilities automatically. However, they target only specific types of vulnerabilities, e.g., taint-style vulnerabilities [106, 191] or missing-check vulnerabilities [164]. On the contrary, our approach is agnostic to specific types of vulnerability.

Code Similarity-based. These approaches [73, 82, 91, 130, 146] start by dividing the entire program into code fragments, which are then converted into abstract representation, e.g., tokens [73, 82, 146], trees [79, 130], or graphs [91, 130]. Lastly, those abstracted code fragments are used to infer similarities between them. The main advantage of this kind of approach is that it can find similar occurrences of vulnerabilities from a single instance. However, these approaches heavily rely on the fact that two semantically similar code fragments are not significantly different from a syntax perspective, which could be false. Instead of finding vulnerable patterns, which cannot be modeled exhaustively, our approach focuses on the conditions that lead to vulnerabilities.

ML & Metric-based. Prior work [18, 24, 67, 69, 70, 93, 94, 104, 137, 143, 189] attempted to apply various machine learning techniques for vulnerability prediction at the source-code level. In addition, other work [25, 54, 111, 116, 117, 148, 152, 193, 194, 198] combined machine learning with software quality metrics, code churns, or token frequency metrics to identify vulnerable code. Simultaneously to our work, a recent paper [40] leverages complexity and vulnerability metrics to identify potentially vulnerable functions. BRAN is different from these works as BRAN considers repository metadata features in addition to code metrics features, which, as we showed in our experiments, significantly enhances prediction capabilities.

Repository-based. There is a line of research investigating the correlation between repository metadata and software defects [107, 108, 116, 126]. Among them, VCCFinder [126] is the most similar work compared to BRAN. VCCFinder tries to locate vulnerability-contributing commits (VCC) incrementally by learning from code metrics and repository meta information within the commit with SVM classifiers. Unlike VCCFinder, BRAN operates on functions as the basic analysis unit. By doing this, BRAN is able to correlate different commits instead of focusing only on one commit at a time. Also, the design goals of the two systems are different. VCCFinder is designed to be more precise while BRAN is designed to achieve higher recall.

2.7 Conclusion

In this chapter, we discussed the need for methods and tools aimed at assisting real-world security testing of large code bases by predicting potentially vulnerable portions of code. To do so, we presented a novel approach that analyzes code with both code metric features and repository metadata features. We implemented our approach in BRAN, a tool that is able to highlight potentially vulnerable functions in large code bases. BRAN relies on CVE reports and machine learning techniques to train classifiers that are able to identify bug symptoms, which capture patterns from both the code and the software development procedure of the project. By evaluating BRAN against four large code bases, we showed

that it is able to predict unseen vulnerable functions effectively. Experiment results also showed that although learned bug symptoms have a certain level of ability to transfer from one code base to another, it is the best practice to use learned bug symptoms from the same project. Finally we leveraged BRAN's results to guide a Linux kernel fuzzer, discovering 13 zero day bugs that would have not been discovered without BRAN's guidance.

Table 2.3: Results of guided fuzzing (Section 2.4.6).

Type of crash	Number of crashes	
	syzkaller	syzkaller _{bp}
Arbitrary memory write	0	16
Arbitrary memory read	2	5
Null-pointer-deref	0	1
Use-after-free	0	4
Total	2	26

Chapter 3

Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

3.1 Introduction

Decentralized Finance (DeFi) drives a revolutionary paradigm shift in the financial landscape. Unlike traditional financial systems that rely on centralized authorities (e.g., banks), DeFi leverages blockchain technology to establish a transparent and decentralized financial ecosystem [135]. Ethereum is one of the most widely used blockchains (with a market cap of 450 billion USD at the time of writing [195]) and arguably the most popular environment for deploying DeFi applications. Hundreds of DeFi applications have been developed on Ethereum to provide financial services with a Total Value Locked (TVL) of almost 82 billion USD [37]. At their core, DeFi applications (or *protocols*) are imple-

mented through one or more *smart contracts*. These smart contracts are immutable programs stored on the blockchain as bytecode and executed by the Ethereum Virtual Machine (EVM) [183]. For instance, a user may request to borrow a certain amount of cryptocurrency from a DeFi lending protocol and provide collateral in a different cryptocurrency as a security deposit. To fulfill this request, the protocol might involve several contracts, such as the “risk-manager contract” (to check for the safety of the borrower’s position), the “oracle contract” (to report the correct price for the exchanged digital goods), and the “vault contract” (to transfer the funds).

While DeFi holds substantial promise, its proponents and users have learned a harsh lesson: Like all software, smart contracts have bugs, and these bugs can be (and have been) exploited by attackers to launch attacks that directly steal money from DeFi protocols [31, 32, 150]. At the time of writing, DeFi hacks have resulted in financial losses of more than 5 billion USD [20, 168]. Some of these stolen funds are known to sustain the activities of criminal groups [50] and dangerous nuclear programs [80]. A smart contract can be affected by multiple types of vulnerabilities [203]. Some of these vulnerabilities are basic implementation flaws (such as integer overflows and reentrancy). These bugs are fairly well-understood by the security community, and in recent years, both academia and industry have proposed a plethora of security tools to identify them [15, 16, 34, 53, 63, 66, 86, 127, 141]. Unfortunately, despite the use of these tools to secure smart contracts, at-

tacks continue to occur. One reason is that certain bugs are tightly coupled to the business logic of individual DeFi protocols. The identification of such logic bugs requires a deep understanding of the protocol’s functionality, which is beyond the reach of most existing automated analysis systems [203].

An alternative approach to identifying (and fixing) bugs is to detect attacks against vulnerable contracts after deployment. To this end, researchers have proposed systems that monitor blockchain transactions. Some systems examine the behavior of transactions that have already been executed with the idea of identifying attacks *a posteriori* [161, 184]. Other systems aim to detect attack transactions before they are committed to the blockchain (while they are still waiting in the public queue) [55]. When such pending attacks are detected, one can attempt to preempt (front-run) the attack transaction with a “defensive” transaction that rescues the funds held by the victim [133, 185, 201].

There are two main approaches for detecting attack transactions: *pattern-based techniques* and *anomaly-based techniques*. Pattern-based techniques leverage rules or heuristics that are applied to the execution traces of transactions [134, 199]. For example, DeFi-Ranger [184] introduces money-flow patterns that capture symptoms of price manipulation attacks. These techniques are generally precise and tend to produce few false alerts, but they are limited to detecting attacks that match known patterns. A second set of detection systems relies on anomaly detection [55, 124]. Their goal is to identify transactions that

result in execution traces, data flows, or other aspects that are sufficiently different from those of benign transactions. Anomaly-based techniques are vulnerability-agnostic and can detect attacks that produce previously unseen anomalous behaviors. However, existing systems often lack explainability and typically generate a substantial number of false positives.

In this chapter, we present HOUSTON, a novel anomaly detection system that performs real-time identification of attack transactions that target DeFi applications on Ethereum. Our key insight and contribution is the automated extraction and modeling of *semantically meaningful signals* from protocol executions. That is, we analyze transaction execution traces and focus on behaviors that reflect the core functionalities of a DeFi application. By doing so, we establish a robust baseline for normal behavior, enhancing our ability to detect outliers that are more likely to be attacks.

We model the behavior of a DeFi protocol by analyzing how functions within the protocol are historically invoked and how the persistent storage of the protocol’s contracts is manipulated. More precisely, our system comprises two distinct models, which are built and continually refined by monitoring live transactions involving the protocol under analysis: The *Interaction Model* and the *Invariant Model*. Here, we use the term “model” not in the machine learning sense, but to describe two purpose-built components: The *Interaction Model* is a control-flow-centric model that captures patterns of function invocations

that perform sensitive operations. The *Invariant Model* is a data-state-centric model that infers relationships, in the form of *likely invariants*, between the inputs of smart contract functions and the values of the permanent storage of the contract. During live monitoring, HOUSTON continually consults and updates each model to examine the incoming stream of transactions. If a transaction is flagged as anomalous by any model, it is then escalated for evaluation by human experts or passed to automated response systems to mitigate the attack (e.g., by front-running the transaction [133, 185, 201]). Transactions deemed non-anomalous are instead utilized to refine the models, enhancing their accuracy and reducing the likelihood of future false positives. Importantly, assessments made by the models are executed rapidly, concluding well within the block generation time, before the possible inclusion of the attack transaction in the blockchain.

In summary, this chapter makes the following contributions:

- We introduce models to capture the normal behavior of DeFi applications and leverage them to quickly detect *when* and *how* a DeFi protocol is under attack.
- Based on our models, we develop HOUSTON, an anomaly-detection system that can be *automatically* tailored to protect *any* given DeFi protocol on Ethereum.
- We compile 115 DeFi protocol incidents into a comprehensive benchmark, drawing from public datasets and incident reports. For each attack, we provide the address

of the contract being attacked, the block and date of the incident, and the hash of the attack transaction. To the best of our knowledge, this is the most precise publicly available dataset on DeFi attacks.

- We compare the results of HOUSTON (94.8% true-positive rate, 0.16% false-positive rate) against state-of-the-art systems and demonstrate its superior performance.
- We deploy HOUSTON as a real-time monitoring solution and evaluate its performance while monitoring 20 DeFi protocols, showing that real-time analysis is achievable with modest hardware requirements.

3.2 Background

3.2.1 Smart Contracts & DeFi

Smart Contracts The Ethereum blockchain supports executing programs known as *smart contracts* directly on-chain. Every smart contract is associated with a unique address, and its deployed code is immutable. Anyone can send a transaction to a smart contract by paying a small (gas) fee. When a transaction is sent, the contract's bytecode is executed by the Ethereum Virtual Machine (EVM) [46]. Smart contracts are typically written in a high-level language, such as Solidity [166], and then compiled into EVM bytecode.

Storage Layout During execution, contracts use two types of data storage: persistent storage and volatile memory. Volatile memory is a form of temporary data storage that exists only for the duration of a single transaction, enabling short-lived data manipulation without incurring the high costs of on-chain storage. On the other hand, persistent storage serves as the contract’s long-term memory, retaining the values of the state variables across transactions [157].

Function Calls To provide any functionality, a smart contract must expose one or more functions that are publicly callable. These functions can be invoked by a transaction that is initiated by a blockchain user (i.e., an externally owned account) or by another contract. When invoking the function of a smart contract, the caller specifies an input (CALLDATA), which contains the function identifier and its corresponding arguments. This enables the compiled contract to direct the execution to the appropriate function.

DeFi Protocols DeFi Protocols provide financial services, such as lending, borrowing, and trading of crypto assets. Examples of DeFi protocols on Ethereum include MakerDAO [103] (a credit platform) and Uniswap (a token exchange) [174]. DeFi protocols are implemented with one or more interconnected smart contracts, each designed to handle specific tasks. For example, at the time of writing, MakerDAO comprises ~415 smart contracts.

3.2.2 Anomaly Detection

Anomaly detection is an analysis approach that focuses on identifying items (or patterns) in a dataset that deviate significantly from the norm [21]. Anomalies may indicate issues varying in severity, ranging from full system compromise to simple unexpected edge cases. For example, if a smart contract sells digital artwork, (benign) execution traces will show that the code *always* checks to ensure that payment has been received before it transfers away ownership of the artwork. In this example, a possible anomaly would be identifying, within a trace, an ownership transfer *without* a check for payment. Anomaly detection has been successfully used for numerous security applications in different fields [85, 165].

3.3 Motivation

HOUSTON helps administrators of DeFi protocols to protect their smart contracts against attacks. An attack consists of one or more malicious transactions that an adversary uses to trigger unintended functionality in a DeFi protocol, typically with the intent to steal assets or disrupt its operations. Frequently, an attack will exploit some smart contract vulnerability, either a basic implementation bug or a logic flaw, to perform actions that the protocol developers did not expect.

HOUSTON is an *off-chain* monitoring solution that provides *real-time* attack detection by continuously monitoring transactions and identifying anomalies. Our system can analyze both transactions that are pending (in the *mempool*) and confirmed (appear in a block). The mempool is a publicly observable queue of transactions that have been submitted to the blockchain but have not yet been included in a block. If an attack transaction is submitted to the mempool, HOUSTON can simulate the execution of the transaction before it is confirmed, and raise an alert if an anomaly is detected. This alert can be used to prevent the attack: For example, administrators could implement an automatic emergency pause mechanism triggered by the alert [205], or the alert could drive automated systems that front-run suspicious transactions [133, 185, 201].

Unfortunately, transactions can bypass the mempool by using a private transaction relay, which forwards transactions to block producers in secrecy [3]. These transactions are generally referred to as *private transactions*, as they are first observed by the public only when they are announced as confirmed in a block. While originally designed to improve the privacy of the blockchain, private transactions are commonly abused to launch stealth attacks [101]. In these cases, our system cannot detect the attack before it is included in a block without cooperation from the private relay. However, even in such cases, an anomaly detection system can provide value. For instance, the attack against the Revest protocol [12] was carried out with four different private transactions (sent in two differ-

ent blocks) over ~ 17 minutes. HOUSTON can detect the first attack transaction (included in a block) and alert the protocol administrators. Thus, if HOUSTON had been deployed, the administrator would have had time to react and (potentially) mitigate the subsequent attacks.

Threat Model DeFi is sometimes referred to as the “Wild West of Crypto” [27]. This is because the ecosystem is plagued by many threats and risks, including hacks, scams, and crypto-wallet thefts. However, different threat categories require distinct identification approaches. In this chapter, we design an anomaly detection system for the real-time identification of attacks (adversarial transactions) that target any *code defect in smart contracts* – such as improper access control, reentrancy, undefined behaviors, and even protocol-specific logical bugs. Consider, for example, the attack against the `Euler` protocol [28]: A logic error in the function `DonateToReserve` allowed the attacker to artificially inflate the conversion rate between the borrowed and collateralized assets, leading to the theft of approximately 200 million USD. In §3.6, we show that HOUSTON detects this and other similar attacks.

We consider out-of-scope other threats whose root cause is related to malicious administration (i.e., rug-pulls [158]), loss of private keys [29], or loss of administrative control due to a governance takeover [76]. These threats generally involve actions performed by entities (e.g., protocol administrators) with a *high level of privilege* over the protocol. As a

result, an anomaly detection system has limited efficacy in addressing such issues, because past actions that were considered “normal” (e.g., updating of an administration key by the protocol administrator) can now turn out to be malicious. Finally, HOUSTON is not designed to detect MEV activities (e.g., *front-running/sandwich attacks*), which are typically attacks that *do not* target code defects in smart contracts.

Technical Challenges Our core challenge is to design an attack-detection tool for Ethereum transactions that achieves both *generalization* and *precision*, while being *real-time-capable*.

(1) Generalizability across attack types with precise detection: Existing detectors often rely on manually crafted, attack-specific rules or on machine-learning features with no clear semantic meaning (e.g., high-dimensional embeddings), limiting both adaptability and interpretability. HOUSTON instead derives vulnerability-agnostic signals from automatically mined invariants and control-flow patterns intrinsic to each protocol. These protocol-specific and semantically meaningful signals produce precise, interpretable alerts, facilitating efficient alert triage and maintaining low false-positive rates over long-term monitoring. (2) Real-time operation on live transaction streams: Detailed data- and control-flow analysis must run within the brief window of block production. HOUSTON achieves this through a lightweight, incremental design that continuously processes traces, detecting exploits in near real-time with bounded computation and memory overhead.



Figure 3.1: *TicketMonster's* Web2 frontend.

3.4 Running Example: TicketMonster

To help demonstrate our system, we introduce a hypothetical DeFi protocol that sells event tickets. Listing 3.1 presents the code snippets of the protocol. *TicketMonster* provides a Web2 user interface (Figure 3.1) that allows customers to buy event tickets (one per transaction) at a given USD price using stable coins: USDC or USDT. For simplicity, every ticket has a fixed price of \$224, with a surcharge of \$500 for VIP service.

Listing 3.1: *TicketMonster's* backend handling ticket sales.

```

1 struct Ticket { uint id; bool vip; }
2
3 contract TicketMonster {
4     bool is_open; // slot 0x0
5     address admin; // slot 0x0
6     uint basePriceInUSD; // slot 0x1
7     uint vipTicketSurcharge; // slot 0x2
8     uint ticketId; // slot 0x3
9     mapping(address => mapping(uint => Ticket)) userTickets; // slot 0x4
10    mapping(address => uint) balances; // slot 0x5
11
12    // ... Bug(A): wrong visibility for function
13    function setVipTicket(address user, uint ticketId, bool isvip) public {
14        // ... logic to handle tickets upgrade
15        userTickets[user][ticketId].vip = isvip; }
16
17    function buy(address token, bool isvip) public returns (uint) {
18        // ... Bug(B): missing address validation
19        IERC20 _token = IERC20(token);
20        uint totUSD = basePriceInUSD;
21        // apply VIP rate
22        if (isvip) totUSD += vipTicketSurcharge;
23        // call to oracle to get ticketPriceInTokens
24        uint amount = getTokAmount(token, totUSD);
25        require(_token.balanceOf(msg.sender) >= amount, "Insufficient balance");
26        require(_token.transferFrom(msg.sender, address(this), amount), "Transfer failed");
27        ticketId += 1;
28        Ticket ticket = Ticket(ticketId, false);
29        userTickets[msg.sender][ticket.id] = ticket;

```

Chapter 3. Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

```
30 setVipTicket(msg.sender, ticket.id, isvip);  
31 balances[msg.sender] += 1;  
32 return ticketId; }}
```

In a typical use case, customers purchase tickets using the website: They would link their crypto wallet (e.g., Metamask [5]), and submit requests to the Web2 website, which, in turn, generates a transaction for the Web3 backend. Importantly, while the transactions generated by the website can be considered safe (i.e., crafted by the developer’s code), nothing prevents more sophisticated users from sending requests directly to the smart contract, bypassing any constraints enforced by the Web2 UI. It is easy to note how the Web3 backend contract fails to enforce certain assumptions made by its frontend UI.

Bug (A) The function `setVipTicket` (Line 13) has the wrong visibility: It is supposed to be `internal` rather than `public`. Because of this, attackers could first buy non-VIP (basic price) tickets and then upgrade with no surcharge by directly calling `setVipTicket`.

Bug (B) The second oversight is the absence of validation for the `token` parameter of function `buy` supplied by users (Line 17). An attacker can simply deploy a malicious ERC20 token contract and use the fake token (instead of USDC or USDT) for payments. More specifically, when the *TicketMonster*’s contract calls to `balanceOf` (Line 25) and `transferFrom` (Line 26), it interacts with the attacker’s token, thus the execution happens according to the attacker’s code. This allows attackers to manipulate the return values of these calls and make the purchase succeed without actually paying [162]. Either bug,

if exploited, can cause severe financial damage. In the following sections, we show how HOUSTON can identify attack transactions exploiting these bugs.

3.5 System Design

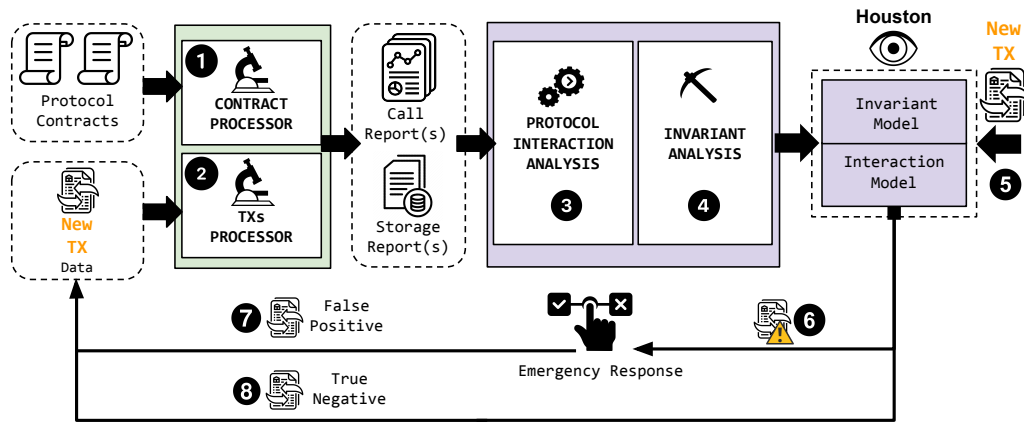


Figure 3.2: HOUSTON Overview. The analyses described in ①-②-③-④ are used to provide data to the models employed by HOUSTON. In ⑤, a new transaction is ready to be examined by HOUSTON. In ⑥, a transaction is considered to be anomalous, and it is sent to an emergency response system. If the transaction is deemed to be a false positive, it is added to the corpus ⑦. HOUSTON will start to generate a new knowledge base automatically. Transactions that do not raise any alert ⑧ are also added to the corpus to update HOUSTON’s models.

Our goal is to design an anomaly detector that identifies attack transactions by monitoring contract executions. Instead of relying on predefined attack patterns or entity reputation, the system extracts high-quality, protocol-specific signals from execution traces to produce timely, accurate alerts. Given a DeFi protocol, HOUSTON builds and continuously updates customized anomaly-detection models as part of its ongoing monitoring process. These models are used to evaluate new transactions directed to the protocol. If

an anomaly is detected, the system raises an alert. Figure 3.2 presents an overview of our system and outlines its operational stages. In the following paragraphs, we describe each of these stages.

① **Contract Processor.** First, we compile a list of all smart contracts that belong to a protocol. When HOUSTON is deployed, we assume that this list can be easily provided by administrators. For our experiments, however, we developed an approach to determine this list ourselves (see §3.6 for details). Then, given the set of smart contracts that compose a protocol, we examine the contracts’ source code and use the Solidity compiler toolchain to extract relevant metadata, such as their public functions, argument types, and storage layouts [157]. We assume that our system typically has access to the contracts’ source code, since HOUSTON users are supposed to be the protocol’s administrators (or developers).

② **Transactions Processor.** Given the output of the previous step and a transaction that interacted with the protocol (i.e., a transaction that triggered the execution of at least one of the contracts in the protocol), we first collect a detailed execution trace of the transaction. Then, we process this data and extract the necessary features that form the basis for our detection models (see §3.5.2 for details).

③ **Protocol Interaction Analysis.** The first detection model (*Interaction Model*) leverages the sequences of function calls. Intuitively, a protocol’s control flow reflects its core actions and their ordering. However, we are not interested in *all* calls. Instead,

we focus on externally initiated calls to the protocol’s public interface that either modify persistent storage or trigger key external operations such as token transfers.

④ **Invariant Analysis.** For our second model (*Invariant Model*), we focus on the values that are *written* to persistent storage and those passed as arguments to function calls. Specifically, we establish “normal” patterns for these values and their inter-relationships. We believe that the semantics of contracts can be meaningfully captured by the values of their state (storage) variables, their function arguments, as well as the relationship between these variables. To do this, we employ likely-invariant inference techniques [43] to capture important protocol properties automatically.

⑤ **HOUSTON Anomaly Detection.** When a new transaction (that targets the protocol) is spotted, HOUSTON (locally) runs the transaction and analyzes the resulting execution trace. Specifically, the two models independently examine various aspects of the transaction trace, including the involved smart contract functions, their argument values, and their validity against the invariants identified in ④. The outputs from both models are then combined to determine a final verdict. If the transaction is found to be anomalous, we raise an alert.

⑥ **HOUSTON Alerts.** Whenever an alert is raised, it can be evaluated by a human analyst or used as a trigger by an automated system to initiate a response. In the event of a confirmed attack (a true positive), administrators can initiate incident response procedures,

such as an emergency pause of the protocol [205]. Alternatively, the transaction can be front-run by one of the recently proposed systems that aim to rescue funds [133, 185, 202]. Conversely, if the transaction was erroneously flagged as anomalous (a false positive), it will be incorporated into the transaction corpus ⑦. The system will automatically update its models, eliminating the recurrence of similar false positives in subsequent analyses.

⑧ **HOUSTON Continuous Updates.** If HOUSTON’s models do not raise any alert for a transaction, HOUSTON automatically adds this transaction to its baseline corpus and updates its models with this new information.

3.5.1 Contract Processor

Recall that most DeFi protocols consist of multiple smart contracts. When protecting a protocol, our system monitors (and takes as input) all the smart contracts associated with this protocol. These contracts constitute the protocol’s footprint on the blockchain. For each contract, we extract its ABI [156], as well as determine its storage layout [157]. These metadata are needed to extract (1) the typed function’s arguments from the raw `bytes` of a transaction’s `CALLDATA`, and (2) to obtain precise information regarding which storage variables are modified during a contract’s execution. To extract these data, we use the off-the-shelf contract’s ABI decoder [156], and a custom procedure named *Storage Variables Decoder (SVD)*.

Storage Variables Decoder We design this procedure to extract precise modifications to state variables. More specifically, the SVD takes as input an arbitrary slot id, the `StorageLayout` of a contract, and a description of the `KECCAK256` operations (i.e., `KeccakInfo`) executed during a transaction (available after tracing a transaction in ②) and returns fine-grained information about the variables stored in a given slot. This process is necessary because variables in the EVM can be packed together in the same storage slot [157]. Without this distinction, proper annotation and invariant calculation would be impossible. For instance, in *TicketMonster* (Listing 3.1), the slot id `0x0` contains a Boolean value (i.e., the variable `is_open`) at offset 0 and an address (i.e., `admin`) at offsets `1→21`. Computing $SVD(0x1, \text{StorageLayout}, \text{KeccakInfo})$ results in $\{v1 = 1, v2 = 0xADMIN\}$. Note that we need `KeccakInfo` to only resolve accesses to dynamic slots (e.g., slots `0x4` and `0x5` in Listing 3.1). For additional details see Appendix 3.10.A.

Binary-only Contracts Currently, if a protocol includes some binary-only contracts, HOUSTON simply avoids calculating the likely invariants in ④ for those contracts. However, these contracts are still considered for the *Interaction Model* in ③. We discuss the impact of source code availability in §3.7.

3.5.2 Transaction Processor

We employ an Ethereum archive node [90] to trace a transaction relevant to a protocol – i.e., one that includes any direct or indirect calls to the protocol’s smart contracts. Specifically, we design a custom tracing plugin to obtain information about the following opcodes: `SSTORE`, `CALL`, `DELEGATECALL`, `STATICCALL`, and `KECCAK256`.

SSTORE For `SSTORE` operations, our custom tracer collects the following information: The slot id being written, the prior value of the storage slot, the new value, the identifier of the function in which this operation happens, the address of the contract executing the opcode (i.e., the code address), and the address of the contract whose storage is being referenced.

CALL When executing a call (`CALL`, `DELEGATECALL`, or `STATICCALL`), we extract the sender address, the destination address, and the raw bytes of the `CALLDATA`.

KECCAK256 The `KECCAK256` opcode applies the `KECCAK256` hash function to its input (the *pre-image*) and produces as output a hash value (the *image*). It is commonly used to compute the address of elements within lists and mappings. For example, consider the mapping `balances` stored at slot id 5 (Line 10, Listing 3.1). To retrieve the balance of a user (i.e., purchased tickets) with key `USER`, the EVM will compute `KECCAK256(USER.5)` and use the result as an address to access the contract’s storage. When tracing a transaction, we record all the pre-images (inputs) for all `KECCAK256`

opcodes. This information allows us to determine whether writes to different memory locations (i.e., slot IDs represented by different KECCAK256 images) target the same dynamic variable (e.g., a mapping storing users' balances).

For a given transaction, the *Transaction Processor* outputs a *Call Report* and a *Storage Report*. The *Call Report* contains *all* the calls performed by *all* the contracts involved in the transaction execution. The arguments for each function call are annotated by the contract's ABI decoder. The *Storage Report* contains all the available information on `SSTORE` and `KECCAK256` operations, as annotated by the *SVD* procedure. In particular, *SVD* refines the accessed slot id into the corresponding variables that are modified (in case there are multiple variables packed into a slot). As a result, the *Storage Report* contains detailed information on every variable modified in the storage of any contract in the protocol. For instance, in *TicketMonster* (§3.4), we would annotate a `Ticket` purchase with its corresponding `id` and `vip` flag.

3.5.3 Protocol Interaction Analysis

The sequence of function calls executed by the protocol's smart contracts (throughout a transaction) provides valuable insights into a protocol's behavior. Intuitively, if a sequence of function calls deviates from typical and common patterns, it is an outlier, and, often, these outliers are attack attempts [55]. However, the entire call sequence of executed

functions (within a transaction) can be very long (e.g., hundreds of nested function calls), may include activities unrelated to a monitored protocol, or may contain uninteresting protocol actions (i.e., a call to a function that simply returns the balance of a user). This can add noise to a model and lead to false positives. Thus, unlike prior work that inspects all function calls, we focus *only* on “critical” function calls that are relevant to the monitored protocol. We name the list of critical function calls *Protocol Interaction*. Our *Protocol Interaction* analysis unfolds in four steps:

(1) Call Direction Identification The analysis first categorizes all calls (that are part of a transaction trace) according to the “direction” of the call. Specifically, we only keep *incoming calls*, that is, calls from a smart contract (or externally owned account) whose address is *not part* of the protocol, to a smart contract *within* the protocol. By retaining only incoming calls, we construct protocol interactions that account for actions initiated by entities external to the protocol. This significantly diminishes the noise caused by outgoing and internal calls (see our ablation study in Appendix 3.10.C).

(2) Critical Call Identification Based on the execution tree provided by the *Call Report*, and the storage writes reported in the *Storage Report*, this analysis marks any incoming function call as *critical* if either the called function itself (or any other function within its call tree), (1) triggers storage changes (performs an `SSTORE` operation) in any of the smart contracts belonging to the DeFi protocol or (2) initiates the execution of a function

matching a predefined list of identifiers. Currently, this list includes four standard ERC20 token [47] functions: `transfer`, `transferFrom`, `safeTransferFrom`, and `approve`. Intuitively, these functions are important because they capture token transfer and approval mechanisms, thereby capturing the movements of digital assets.

(3) Direct ERC20 Token Operation Filtering Many DeFi protocols include one or more ERC20 token contracts. External users (and contracts) can directly interact with these token contracts, performing operations (e.g., `transfer`) without involving the rest of the protocol’s logic. These operations tend to occur in large volumes and generally do not correlate to potential attacks but rather add substantial noise to the model. Therefore, HOUSTON filters out all *incoming* ERC20 calls from the protocol interaction pattern. Importantly, we only exclude *incoming* ERC20 operations (e.g., `approve` in Figure 3.3). If a call to a non-ERC20 function triggers downstream token movements (e.g., a call to `transfer`), HOUSTON still considers it as a part of the protocol interaction, e.g., in Figure 3.3, the *incoming* call to function `pay` triggers an ERC20 `transfer`, thus, it is part of the final fingerprint.

(4) Interaction Pattern Identification We use the sequence of critical function calls as a *fingerprint* of the transaction. Specifically, from the *Call Reports* of historical transactions, the *Protocol Interaction Analysis* builds a database of all observed interaction fingerprints (later used by the *Interaction Model*). For example, in *TicketMonster* (§3.4),

as benign users buy tickets via the Web2 UI, the protocol interaction database would contain only one fingerprint: `[buy]`. The reason is that neither the call to the price oracle (Line 24) nor the call to function `setVipTicket` (Line 30) is an *incoming call* (i.e., a call coming from outside the protocol and directed to it), thus, they are not considered in the final fingerprints. To better illustrate our algorithm, Figure 3.3 shows a more complete example of how we reduce a complex sequence of calls (in and out of a protocol) to the signature `[borrow, pay]`. By focusing on critical functions, we significantly reduce the complexity of call sequences while preserving the relevant information regarding the protocol’s function calls. For example, in the `Euler` protocol attack, the attack transaction made 150 calls, whereas the *Protocol Interaction* representation of the attack comprises only 8 calls. These 8 calls outline the exact steps of the attack as security experts described in their writeups [136]. In our experiments (§3.6), HOUSTON successfully detects the attack transaction with the help of these fingerprints. Moreover, we show that the focus on critical functions (rather than modeling the entire function call trace) is critical to the accuracy of this model (Appendix 3.10.C).

3.5.4 Invariant Analysis

Program Invariants Invariants represent properties within a program’s code that always hold at a program point or a set of program points [43]. Commonly, identifying program

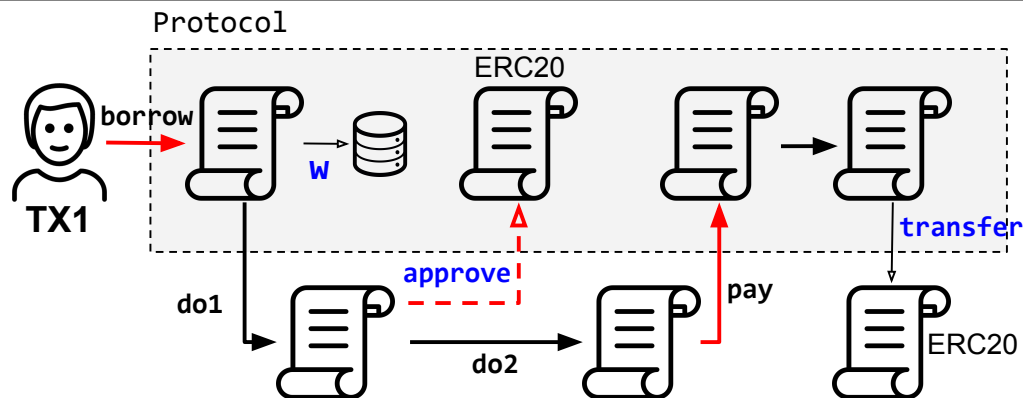


Figure 3.3: Protocol Interaction Analysis. The protocol’s smart contracts are in the gray area. Function `borrow` triggers a storage write in one of the protocols of the contract. Function `approve` is filtered out. The function `pay` leads to an ERC20 transfer. The fingerprint of TX1 is `[borrow, pay]`, but we save function selectors instead of their names.

invariants requires static analysis techniques, such as symbolic execution and abstract interpretation [149]. However, those techniques suffer from scalability limitations when used to analyze complex contracts. Moreover, for our purposes, we are not interested in proving that certain program invariants always hold. Instead, our focus is on identifying relationships between variables that consistently hold in incoming transactions, reflecting the intended behavior of a smart contract’s execution. These relationships, when violated, may indicate an attack. Hence, for HOUSTON, we collect *likely invariants* [43].

Likely invariants are properties that are dynamically inferred from values observed during multiple program executions. For instance: in *TicketMonster*, the argument `token` has been historically observed to be equal to `0xUSDC` or `0xUSDT` (these are the only two currencies allowed for payment by the Web2 UI, Figure 3.1). This dynamic inference approach generally produces relevant relationships, but it comes at the cost of soundness

(that is, a likely invariant may not hold true for all executions). Nevertheless, given enough observations of a likely invariant, the confidence of the observed property increases [44]. The goal of the program invariant analysis is to automatically identify likely invariants at the function boundaries of a smart contract. Specifically, we infer relationships between a function’s inputs (its arguments and initial storage state) and its outputs (represented by the resulting storage state). Unlike return values alone, storage updates capture meaningful behavioral effects because persistent state changes influence both the current execution and future interactions.

Likely-Invariant for Attack Detection Likely-invariants naturally arise from the execution of DeFi protocols. Our core intuition is that likely-invariants can be used to identify attacks against a protocol, at the cost of tolerating a small percentage of false positives (i.e., “benign anomalies”). Intuitively, since attack transactions are intrinsically abnormal (i.e., they exploit edge cases in the protocol), they are likely to breach one of these likely-invariants, and, therefore can be observed.

Likely-Invariants Class Selection The selection of likely-invariant classes requires considering a wide range of possibilities. Intuitively, mining *all* possible unary and binary relationships among *all* the variables in a smart contract would rapidly make the problem intractable (e.g., the Daikon [43] engine offers more than 200 invariant classes). To reduce the number of invariant classes, we initially experimented with a broad range of them on

a pilot dataset of 16 DeFi incidents covering diverse attack types. While experimenting with different invariants on our pilot dataset, we observed several practical challenges: Some invariant types were overly sensitive and triggered frequent false positives (e.g., the `UpperBound` invariant, which alerts whenever a variable exceeds its historical maximum); others (e.g., ternary invariants) led to a combinatorial explosion in run-time or lacked clear semantic interpretation. We iteratively filtered out such invariant classes and focused on those that both: (1) showed violations during real attacks, and (2) had clear, human-interpretable semantics. This trial-and-error process led us to the following four simple invariant classes used in the final system:

(C1) For individual integer variables, the analysis checks whether a variable is *never* zero. An unexpected zero may signal abnormal behavior in arithmetic operations.

(C2) For individual address and bytes variables, the analysis determines if the address or bytes variable always equals a specific value, or if it is drawn from a set of possible values.

(C3) For relationships between two integer variables, the analysis checks if any of the ordering relations ($=, \geq, \leq$) always hold. For example, an invariant may state that the amount of redeemable tokens is *always* less than the user's balance.

(C4) For relationships between two strings, addresses, or bytes, we evaluate whether the two variables are always the same. This invariant type can help when, for example, the

developers assume that two addresses should always be identical to ensure that a transaction originates from the same source, but fail to enforce the assumption.

Selective Mining for Binary Likely Invariants The attentive reader may have noticed that our method for generating likely invariants over the contracts’ variables adopts a “brute-force” strategy. Specifically, HOUSTON checks all possible relationships or properties (of types **C1–4**) over variables and variable pairs as they are observed during the processing of a protocol’s transactions. While this approach is acceptable for unary invariants, it quickly leads to the generation of numerous spurious binary invariants that lack any real semantic meaning (e.g., `timeStamp < reserve`). Although many of these invariants are unlikely to ever be violated, some may introduce false positives. To address this, we precompute all variable pairs in each contract that the *Invariant Model* might compare and use an LLM (GPT-4.1 [121]) to check if each comparison is semantically meaningful in the context of the contract code (see Appendix 3.10.H for the prompts). During monitoring, the *Invariant Model* only mines (and checks) over the variable pairs deemed semantically meaningful. We quantify the effectiveness of this selection step in §3.6.3.

Likely-Invariants Inference To identify likely invariants, we combine the information from *Call Reports* and *Storage Reports* (generated in ②) to create data traces of function arguments and storage variables at the entrance and exit of all function calls. The *Invariant Analysis* then determines, for each invariant category of interest, whether an invariant

over a single variable, or between two (semantically comparable) variables, consistently holds across *all* executions at a specific program point. If this consistency is verified, we record the invariant and the number of transactions supporting it (as a metric of confidence) for future reference. Importantly, our current invariant inference algorithm is *not* designed to extract sophisticated invariants of the kind that a developer might manually specify (e.g., `collateralValue × collateralFactor ÷ debtValue ≥ 1`, which was violated in the Euler incident). Instead, its goal is to mine a large number of *simpler* candidate likely invariants. While some may be quickly violated (and thus deemed unlikely), others capture meaningful properties with broad coverage, emerging as valuable indicators for detecting attacks, either directly or through the side effects of malicious transactions. Our invariant extraction algorithm can be easily extended with more advanced systems [95] if required by the protocol.

3.5.5 Anomaly Detection

For every new transaction, HOUSTON creates an execution trace and consults its models to identify anomalies.

Interaction Model The *Interaction Model* computes the transaction’s fingerprint as described in §3.5.3 and compares it against those in the database. The model reports an anomaly if the fingerprint has not been seen before. For example, if an attacker tries to

exploit *TicketMonster* using Bug (A) (Listing 3.1, Line 13) by first calling the `buy` function and then, in another transaction, calling `setVipTicket` to upgrade their ticket, they would generate an unseen fingerprint [`setVipTicket`].

Invariant Model The *Invariant Model* observes all protocol transactions and computes the likely invariants. Then, it uses such invariants to detect anomalies (invariant violations) in new transactions. In the ideal case, where the monitored application is stable and has seen significant usage, a transaction should be flagged as anomalous if *any* of the computed invariants is violated. For instance, in our *TicketMonster* example, an attacker attempting to exploit Bug (B) by providing a token address other than `0xUSDC` or `0xUSDT` would violate the invariant $token \in \{0xUSDC, 0xUSDT\}$, which is an invariant of category **C2** as described in §3.5.4. For newly deployed contracts or functions that see limited use, the supporting observations for an invariant may be minimal. This scarcity of data can make it challenging to determine whether an inferred likely invariant accurately reflects the core semantics of the contract or whether the invariant is merely incidental due to insufficient transaction diversity, suggesting that alerts on its violation may be groundless. To address this, the *Invariant Model* enforces a waiting period for each likely invariant, allowing more user interactions with the protocol. During this period, any violation will lead HOUSTON to discard the invariant without triggering an alert. The waiting period is defined by two hyperparameters: the minimum transaction count supporting the invariant (N) and the

minimum contract age (T). A violation is within the waiting period only if *both* conditions are met. In this study, we set $N = 10$ and $T = 12$ hours, but our experiments show that detection outcomes are stable across various settings (see Appendix 3.10.D for more details). The final decision is simple: If *at least one* of the two models report(s) an anomaly, HOUSTON triggers an alarm.

3.5.6 Continuous Updates

HOUSTON continuously incorporates new interaction fingerprints and updates invariants, keeping its behavioral models aligned with the protocol as it evolves. This incremental learning is essential to maintaining a low false-positive rate.

Interaction Model Update Whenever a new protocol interaction is observed, HOUSTON raises a warning and then simply adds the interaction fingerprint to its database.

Invariant Model Update This involves three main operations: mining new invariants, discarding any that have been violated, and increasing the confidence of existing invariants by incorporating additional observations. For example, assume the invariant $token \in \{0xUSDC\}$ has been established for *TicketMonster*. If a new transaction uses $0xUSDT$ as `token` (Listing 3.1, Line 18), HOUSTON will raise a warning, discard the previous invariant, and establish the new invariant: $token \in \{0xUSDC, 0xUSDT\}$. To enable continuous updates of the *Invariant Model*, we implemented our own invariant mining en-

gine inspired by `Daikon`, a state-of-the-art system for inferring likely invariants [43]. Our invariant miner adds support for incremental invariant inference, which enables HOUSTON to quickly incorporate new information into its models.

3.6 Evaluation

3.6.1 DeFi Attacks Dataset

To evaluate HOUSTON, we compiled 115 DeFi protocol incidents into a comprehensive benchmark, drawing from both previous work [55, 205] and public reports of security incidents [138, 163]. We curated our dataset to be as comprehensive as possible by including all incidents within the defined scope. Only those incidents that fell outside this scope (as outlined in §3.3) or occurred on non-Ethereum chains were excluded. Our dataset includes protocols of varying application types and exploits against a diverse set of vulnerabilities. Note that each incident corresponds to a distinct protocol. For incidents with multiple attack transactions, we label the *earliest* transaction as the ground truth, as detecting this initial action allows administrators to respond swiftly, for instance, by pausing the protocol to prevent further damage. Notably, this approach is more challenging than merely identifying any one of the attack transactions. For each attack, we report the address of the

contract being attacked, the block and date of the incidents, and the hash of the first attack transaction. Appendix 3.10.B, Table 3.2, summarizes the DeFi protocols in our dataset.

Identify Smart Contracts of a Protocol In §3.5, we described how our system takes as input the addresses of the smart contracts belonging to a DeFi protocol and then uses incoming transactions to build its models. In practice, protocol administrators can (easily) identify the involved contracts. For our evaluation, however, this information is not readily available. We therefore approximate protocol membership as follows. We first identify the deployer of the attacked (victim) contract, i.e., the account that created it. We then include all contracts directly deployed by this account as part of the protocol. Since contracts may also be created through internal transactions, we further inspect the deployer’s internal transactions and include any contracts created in this way.

3.6.2 Experimental Setup

We evaluate our system on a server with dual Intel Xeon Gold 6330 processors, 512 GB of RAM, and 20 TB of SSD storage. To evaluate HOUSTON’s capabilities, we design two experiments: the *Historical Evaluation* and the *Live Performance Evaluation*. The goal of the former is to demonstrate the system’s high precision and recall across 115 historical attacks, while the latter aims to showcase its effectiveness in real-time scenarios when monitoring 20 protocols in parallel.

3.6.3 Historical Evaluation

Given a protocol, we first collect all past transactions that interacted with any of its contracts, up to the point of the first known attack (as per our dataset 3.6.1). HOUSTON then analyzes these transactions in chronological order to identify anomalies. We record whenever an alert is produced, and we stop our evaluation after processing the reference attack transaction. In this evaluation, we consider any alert that does not correspond to the reference attack transaction to be a false positive. While this approach may be imprecise—particularly if there are unknown attacks preceding the reference incident—it provides an upper bound on the number of false positives generated by HOUSTON. It is also worth noting that while an alert from HOUSTON might be considered a false positive, the flagged transaction may still exhibit anomalous features that can offer valuable insights, e.g., a new behavior used in the protocol, or a new contract appeared in the protocol that is interacting with it in new ways. Finally, when evaluating the attack transaction, if HOUSTON flags the transaction as anomalous, we consider it a true positive; otherwise, we consider it a false negative. Our results show that HOUSTON is effective in detecting attacks for a diverse set of protocols. Specifically, out of 115 DeFi protocol incidents in our dataset, HOUSTON detected 109 attacks ($\sim 94.8\%$). HOUSTON also exhibited a low false positive rate of 0.16%. We present a per-protocol results overview in Table 3.6 in Appendix 3.10.I.

True Positives In contrast to detection systems based on neural networks [55], the alerts produced by HOUSTON have a high level of explainability. That is, they directly show which calls are anomalous or which invariants have been violated. This makes it much easier for human analysts to determine the root cause of an attack and to discard potential false positives. This level of explainability enables us to systematically examine how HOUSTON’s detected anomalies align with the actual exploit mechanisms. Specifically, we analyzed all 109 detected incidents and compared our alerts with the corresponding public postmortem reports. For each incident, we examined alerts from both models and used the more conclusive one as its representative signal. Among these incidents, HOUSTON produces *causal* detections in 57 (52%) cases, where the invariant violation or interaction pattern directly pinpoints the root cause of the vulnerability. In these situations, the abnormal behavior exposed by HOUSTON is so fundamental to the exploit that carrying out the attack without triggering such an alert would have been impossible. Another 43 (39%) incidents fall into the *indicative* category, where the detected violation reflects a consequence of the attacker rather than the immediate exploitation of the bug. These alerts often capture secondary effects that reveal the attacker’s intent or correspond to side effects of the exploit. While such signals might be avoided by a highly sophisticated attacker, it depends on the specific circumstances or goals of the attack, and doing so would significantly increase its difficulty. Finally, 9 (8%) incidents are *incidental* detections, where

the triggered signal is unrelated to the exploit and uninformative for triage. Most of these cases are in protocols exploited early in their deployment, and the alerts are raised because the transaction paths are novel.

Below, we discuss two interesting true positives that our system detected (refer to Appendix 3.10.G for more examples).

(1) *RevestFinance*. This attack was detected by both of HOUSTON’s models. Specifically, the invariant violation points to a mismatch between the two storage variables `fnftID` and `fnft_created`, which are supposed to be equal when entering the function `mint` (C3, §3.5.4). This mismatch was caused by the exploitation of a reentrancy vulnerability, and the attack could not have succeeded without triggering this condition (*causal*). For the *Interaction Model*, no previous transaction ever called `mintAddressLock` twice in its protocol interaction (hinting at possible reentrancy) and thus triggered a warning (*indicative*).

(2) *Euler*. Both HOUSTON’s models identified the attack. Specifically, the *Interaction Model* reported a new protocol interaction where a call to `donateToReserves` preceded a call to `liquidate`, which is essential for the exploit to succeed (*causal*). At the same time, the invariant model reported a violation during the execution of `liquidate`. In all prior executions, `deferLiquidityStatus` was *never zero* for `liquidate` (C1, §3.5.4). This violation arose because the attacker deliberately skipped `deferLiq-`

uidityChecks, which is normally invoked by all legitimate users. Although the exploit could still succeed even with this check in place, omitting it reflects a distinct behavioral deviation – an attacker’s willingness to bypass standard safety routines. Therefore, we classify *Invariant Model*’s detection as *indicative*.

False Negatives HOUSTON failed to detect an attack in only 6 incidents out of 115 (5.2%). After manual investigation, we observed that four incidents would require mining additional types of invariants. For instance, for the `nomad` incident [31], it would be necessary to infer invariants on values read from the storage (currently, we focus only on storage writes, as discussed in §3.5.2). Of the remaining two cases: `azukidao` was exploited too soon (4 days) after deployment, and, therefore, HOUSTON simply did not have enough data to learn anything meaningful; `snood`’s exploit directly targeted a buggy reimplementation of a `transferFrom` in the protocol, and since incoming ERC20 calls are filtered to reduce false positives (as explained in §3.5.3), this attack was not detected.

False Positives Overall, HOUSTON had 13,827 false positives out of 8,586,421 transactions evaluated across all 115 protocols over their entire history (false-positive rate 0.16%, 0.4 false positive per protocol per day on average). As is shown in Figure 3.4, 64 (55.7%) protocols have $\leq 1\%$ false positive rate; 92 (80.0%) protocols have $\leq 3\%$ false positive rate. Table 3.6 in Appendix 3.10.I shows the detailed false-positive numbers (and rates)

for each protocol. We investigated these false positives and identified two main reasons for these errors:

(1) *Historical Data Availability.* When there are too few observations (or no observations in the case of new contracts), HOUSTON has very little information to learn from and will suffer from false positives during a short initialization period. Luckily, as transactions accumulate, HOUSTON’s invariants and interaction fingerprints become more reliable, and the false positive rate drops quickly. In Figure 3.5, we show the correlation between the number of transactions available for a protocol and the false positive rate. Notably, *all* of the 23 protocols with a false positive rate exceeding 3% had fewer than 800 total transactions. In practice, developers can mitigate this cold-start effect by bootstrapping HOUSTON with their own test cases or simulation traces prior to deployment. These synthetic transactions provide initial behavioral diversity for the model to learn from, allowing it to establish preliminary invariants and interaction patterns and, thereby, significantly reducing false positives during the early monitoring phase.

(2) *Likely-Invariant Quality.* In §3.5.4, we described how we used an LLM to selectively mine binary invariants to avoid nonsensical pairs. We found that this selection step reduced the number of false positive alerts in the *Invariant Model* by 27.2% (from 10,579 to 7,702), while preserving the model’s detection capability (true positives remain unchanged). Because our filtering is purposefully conservative (see our prompts in Ap-

pendix 3.10.H), some spurious relationships might persist. We manually analyzed a random sample of 100 mined invariants from the *Invariant Model*. Of these, 84 were deemed meaningful: we could imagine plausible scenarios in which violations of these invariants can be indicative of abnormal executions (even if such violations may never occur in practice). Among the remaining 16 that we found to be spurious, two were unary invariants that asserted unjustified restrictions on free parameters. For instance, one such invariant incorrectly assumed that the `spender` parameter in the `allowance` function must belong to a small, fixed set of addresses. The remaining 14 were binary invariants involving variable pairs that are not semantically comparable. This could be further resolved by utilizing a better reasoning model at the cost of a more expensive pre-filtering step (the process cost approximately \$100 of LLM credits across the entire dataset).

(3) *Operation Repetition and Permutation.* The *Interaction Model* drastically reduces the call trace of a transaction to a few, meaningful operations. However, the repetition (and permutation) of such operations can lead to new interaction fingerprints that cause false alarms. For instance, if a user performs five `deposits` and one `withdraw` against a monitored protocol, depending on where the `withdraw` is placed among the five `deposits`, the transaction may generate six unique fingerprints. Regardless, these false alarms appear less frequently as transactions accumulate. To mitigate this, one may associate each operation with the underlying logical entity it acts upon (e.g., a stake or lien).

This entity-level grouping effectively partitions long, flattened parallel sequences into independent sub-sequences, thereby preventing the combinatorial explosion of possible interaction patterns.

Given the statistics and root-cause patterns above, it is useful to consider the operational impact of false alerts. Although HOUSTON produces false alerts on the whole historical dataset, this does not undermine its practicality. Most false positives occur during the initial (learning) phase, after which the rate stabilizes at very low levels, with mature protocols typically seeing fewer than one non-exploit alert per day (see Figure 3.5, Table 3.6). Because alerts are explainable and closely aligned with actual exploit mechanisms, HOUSTON remains effective despite the presence of false positives. To further evaluate the false positive rate of HOUSTON on mature, high-quality protocols without known history of exploitation, we collected all transactions up to the time of experiment (block 20400000, totaling over 13 million) of an additional set of popular Ethereum DeFi protocols with TVL (Total Value Locked) greater than \$500M [37] (see Appendix 3.10.F for the complete list). Then, we performed the same experiment as the one for the DeFi attacks dataset. For this experiment, we reported an aggregate false-positive rate of 0.08%, which is significantly lower than that for the DeFi attacks dataset. This supports our observation that, for well-used protocols, false-positive rates drop significantly, indicating that our system is effective in real-world settings, where accuracy is essential. Finally, to

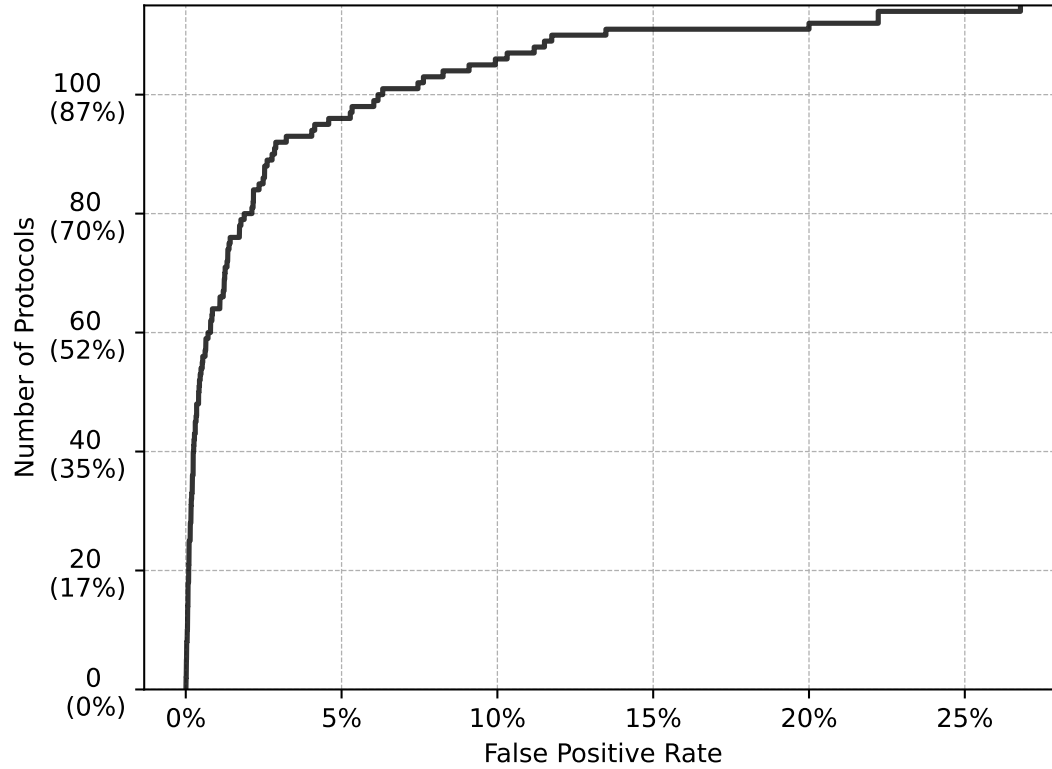


Figure 3.4: Cumulative distribution of false positive rates.

reduce false positives, one could also initialize HOUSTON’s models with existing benign transactions.

3.6.4 Comparison with Existing Systems

APE APE is a system that automatically synthesizes adversarial smart contracts to imitate (and front-run) profitable transactions in real-time [133]. Although HOUSTON and APE pursue fundamentally different objectives (anomaly detection vs. profitability detection), a comparison between the two is still meaningful, as Qin et al. suggest that APE can also

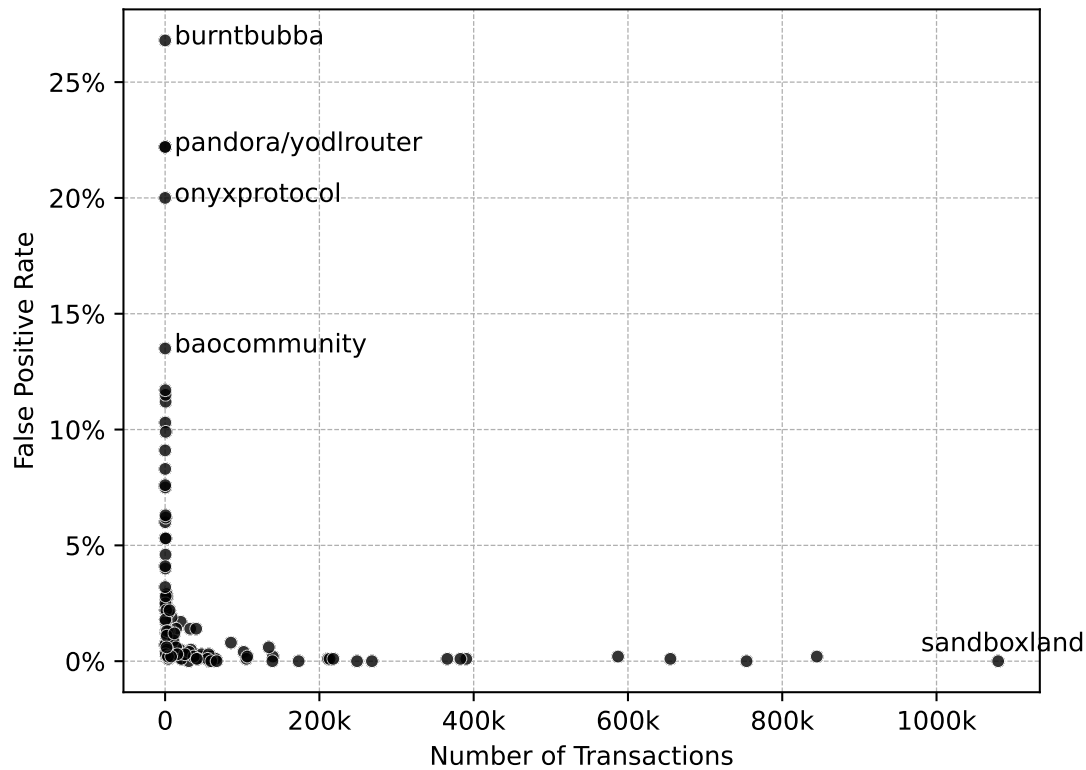


Figure 3.5: Overview of the correlation of false positive rates with the total number of transactions available per protocol. Note that protocols with high false-positive rates are evaluated with an exceptionally low number of transactions. See Appendix 3.10.I Table 3.6 for detailed per-protocol results.

be leveraged as a defensive mechanism. For this comparison, we were unable to run APE directly (the system is not open-sourced); however, the authors provided us with the list of transactions that APE historically replayed on the Ethereum mainnet. This transaction set spans from block 12936340 to block 15253305, overlapping with our dataset. The overlap includes 2.4M transactions, among which there are 20 attacks. Out of these 20 attacks, APE successfully countered 4, while HOUSTON flagged 19 (both systems share a false negative). Within the same reference time frame, APE exhibited a false positive rate of

0.15%, compared to 0.10% for HOUSTON. Theoretically, HOUSTON can enhance APE’s attack capabilities when both target the same DeFi protocol. Upon receiving a transaction, APE synthesizes a frontrun, while HOUSTON checks if the activity is malicious. If confirmed, APE can promptly deploy the crafted transaction.

BlockGPT Gai et al. proposed `BlockGPT` [55], a generic approach for DeFi anomaly detection based on neural networks. At the time of writing, the authors have not yet made their system available as open source. Thus, we chose to compare our results based on the protocol attacks reported in their paper in Figures 5, 10, and 11. Also, the authors did not disclose the exact transaction hashes that `BlockGPT` marked as attacks. Thus, we must assume that `BlockGPT` identified the same transactions as our investigation. While we understand this can create discrepancies in the comparison results, this is our best attempt to make a fair comparison. To compare with `BlockGPT`, we selected the 28 attacks that are both in scope for HOUSTON (i.e., on Ethereum) and present in `BlockGPT`’s dataset. Overall, our system detects 27 attacks across these 28 protocols, whereas `BlockGPT` detects only 15. Our only false negative is 88mph. Unfortunately, it is not clear why `BlockGPT` would detect an anomaly against 88mph because (1), the system naturally suffers from limited explainability, and (2) we have no information regarding the transaction hash labeled as an attack by the authors. Table 3.6 shows a detailed comparison between the detection results for HOUSTON and `BlockGPT`. Since we do not have

access to `BlockGPT`, we cannot run it directly on our dataset to determine its false positives. Instead, we look at the reported results in Figure 4 of the paper. Overall, `BlockGPT` achieves a false positive rate of less than 10% for 58% of the protocols in its scope. As a comparison, `HOUSTON` has a false-positive rate of less than 10% for 92.2% of the protocols. Notably, `HOUSTON` achieves a false positive rate of 1% or less for 55.7% of the protocols.

TXSPECTOR `TXSPECTOR` [199] is a logic-driven framework that detects attacks by analyzing execution traces. It relies on user-defined control and data dependencies rules tailored to specific attack patterns (the original paper covered, for example, re-entrancy, unchecked call, and suicidal vulnerabilities). As a consequence, the range of attack types `TXSPECTOR` can detect is limited. When applied to our dataset, it identified only 25 out of 115 attack transactions (21.7%), whereas `HOUSTON` achieved a true-positive rate of 94.8%. Moreover, not all vulnerability types can easily fit into the framework, making it impractical to rely solely on writing additional custom rules to achieve more generic detection coverage.

DefiRanger The system targets price manipulation attacks on DeFi applications [184] by constructing and analyzing a cash flow tree. It successfully identifies attacks on several protocols in our dataset, as detailed in Section 6.2, Table 5 of their paper. Of the 9 reported attacks, we excluded `Pluto` (Binance chain), `DRC`, and `MET` (missing trans-

actions' hashes). Our evaluation shows that HOUSTON detected all 6 remaining attacks and flagged the correct transaction as anomalous. Additionally, our dataset includes 23 other price manipulation attacks not included in DeFiRanger's dataset, which it could plausibly detect (though this is unverified due to lack of open source). The remaining 86 attacks (74.8%) in our dataset involve other vulnerability classes and would likely be false negatives for DeFiRanger (out of scope).

Transaction Length Baseline It seems to be common wisdom in the blockchain community that it is sufficient to look for abnormally long transactions to detect attacks. While this might be true for some exploits, we argue that transaction length is generally a weak signal. To demonstrate this, we evaluate the transaction-length-based approach by using a simple heuristic that just measures the length of a transaction (in terms of the number of function calls performed) and compares it to the maximum length observed up to that point. We raise an alert whenever a new transaction is longer than any observed before. As expected, this basic approach detects only 46 attacks (40%) out of the total 115.

3.6.5 HOUSTON Live Performance Evaluation

It is essential for anomaly detection systems in DeFi to keep pace with block production. Specifically, in the context of Ethereum, a system must be able to evaluate all the transactions committed to the latest block in less than 12 seconds (*block-time*). Moreover,

to offer attack-prevention capabilities (i.e., identifying attack transactions in the mempool before they are committed to a block), a system must also manage the influx of transactions sent to the mempool. Therefore, it is necessary to test HOUSTON in a real-life setting to demonstrate how it can keep up with both the pace of the live production of blocks *and* the transactions sent to the mempool. To evaluate HOUSTON as a real-time anomaly detector, we deploy it live against 20 randomly selected protocols (from our dataset) that are operational at the time of this experiment (protocols marked with $\star$ in Table 3.6, Appendix 3.10.I). We deployed HOUSTON to monitor new incoming transactions as discussed in §3.5. HOUSTON analyzed *all* mainnet transactions from April 9th to April 28th, 2024 (20 days). This includes transactions in the mempool *and* committed blocks.

Transactions Filtering HOUSTON observed a total of 42.6M transactions. We discarded 50.2% of them either because they perform simple ETH transfers (no smart contract involvement), or because they directly invoke basic functions of well-established token contracts (e.g., `USDC transfer`). This step took a few nanoseconds per transaction, which is negligible.

Tracing Performance To check if a transaction calls the monitored protocols, HOUSTON must replay it to collect a trace. This is because simply checking for direct invocations is insufficient; internal transactions also need to be considered. On average, we observed ~ 13 transactions per second. Given the average tracing time of 3 ms per transaction, a

single *tracing worker* can handle this load in 39 ms when monitoring all 20 protocols simultaneously. Note that a *tracing worker* only requires ~ 150 MB of RAM and 25% of a CPU core.

Processing Performance If a transaction involves *any* of the contracts in a monitored protocol, HOUSTON needs to extract the *Call Report* and the *Storage Report* ②. Overall, during live experiments, we observed a total of 95.6K transactions directed to the monitored protocols. On average, the number of transactions for which this step is required is about one transaction per block. Given our average processing time of 180ms, this load can be handled by one *processing worker* equipped with ~ 1.5 GB of RAM and 1 CPU core.

Detection Performance After processing, HOUSTON performs the following two steps: (1) HOUSTON consults its models to make a detection decision ⑤, and (2), it updates the models with the new transaction’s information ⑦/⑧. Together, these two operations take, on average, ~ 28 ms per transaction. The resources required by a *detection worker* are 2GB of RAM and 2 cores. Each monitored protocol needs at least one dedicated *detection worker*.

Worst Case Performance The average load for HOUSTON to monitor 20 protocols is light and could be handled by one *tracing worker*, one *processing worker*, and 20 *detection workers*. Of course, our system cannot just be built for the average case. We also need

to take into account bursts of transactions. During our live monitoring, we observed the following peak instances: A maximum of 362.9 transactions/second that needed to be traced, as well as 2.2 transactions/second that required further processing. Moreover, we observed individual worst-case processing times of 337ms for the *tracing worker*, 2.9s for the *processing worker*, and 810ms for the *detection worker*. See Appendix 3.10.E for a summary of average and worst-case performance (Table 3.3) and transaction load (Table 3.4). Overall, combining all worst-case instances, HOUSTON would require 121 tracers, 8 processing workers, and 40 detection workers (2 per protocol). Given these requirements, HOUSTON needs ~ 110 GB of memory and 120 cores to keep pace with real-world Ethereum mainnet traffic. These requirements can be matched by a single modern server, such as the one we used in our evaluation, where we never encountered all worst-case instances simultaneously. Importantly, we think this experiment provides evidence of feasible real-world deployment. Should HOUSTON be adopted in an industrial setting, there are clear engineering venues for achieving faster tracing and detection times (e.g., using reth [125] instead of Erigon [90], and re-implementing HOUSTON in Rust [144]).

Live Setup Our live evaluation used 8 tracers, 4 processing workers, and 20 detection workers (one per protocol). The system showed no latency or resource issues: The transaction queues remained empty, memory peaked at ~ 80 GB, and CPU load averaged 8.0 across 112 cores.

Live Monitoring Warnings During the live evaluation, HOUSTON kept low false positive rates. Overall, HOUSTON raised a total of 75 warnings (0.08% of the 95.6K transactions directed to monitored protocols). For 9 of the 20 monitored protocols, HOUSTON never raised an alert. For 10 of them, HOUSTON raised fewer than 10 alerts throughout the 20-day period. The only exception is `conicfinance`, which reported 24 warnings over 901 transactions (2.7%). The reason for this is that a newly introduced `ConicPool` address triggered multiple invariant violations at various program points.

Additional Monitoring We conducted an additional three-week live experiment (October 27th to November 16th, 2025) on the same 20 protocols. Across 108.5K related transactions, HOUSTON generated 71 alerts (0.07%), consistent with the false-positive rate in the previous period. Manual analysis confirmed that none were attacks, and no external sources reported incidents during this period.

3.6.6 HOUSTON for other EVM-compatible chains

To further validate the generality of our approach, we ported HOUSTON to another major EVM-compatible blockchain: *Binance Smart Chain* (BSC [17]). In general, three key components are required to adapt our system to a new EVM-compatible chain: (1) a **native transaction tracer**, (2) a **blockchain index database**, and (3) a **contract analysis tool** capable of fetching and inspecting smart contracts to extract their *storage layouts* and

ABIs. To this end, we ported our Erigon plugin to bsc-erigon [120], we used Dune Analytics [42] to fetch the protocol’s boundaries and related transactions, and we adapted the toolchain in Figure 3.2, Step ①, to work with BSC. For this experiment, we consider the incidents listed on *DeFiHackLab* [163] that occurred on BSC between May and October 2025 (20 incidents over six months). Among these, we discard seven incidents because the vulnerable contracts are available only in binary form, and an additional one due to insufficient information about the vulnerable contract. HOUSTON successfully identifies attacks in 11 of the remaining 12 incidents, with a low false positive rate of 0.19%. The only undetected case (pdz) does not represent a genuine detection failure: HOUSTON instead flags an earlier transaction (0x88f8741d) that occurred 202 days before the labeled incident and exhibited a highly similar price-manipulation pattern. Notably, this earlier attack was never publicly reported; the protocol’s maintainers appear to have been unaware of it and continued normal operation.

3.7 Discussion and Limitations

Source Code Availability The precision of HOUSTON’s models depends on the availability of contract ABIs and storage layouts. Although partial information can be inferred through binary-only analysis [63, 66, 141], accuracy generally degrades. Accordingly, we assume developers using HOUSTON have access to their own source code. However, third

parties analyzing unverified contracts must rely on decompilers to recover ABIs and storage layouts. Notably, recent advances [48] leveraging LLMs have achieved up to 49.8% success in recompiling binary-only contracts, enabling the extraction of ABIs and storage layouts. Practically, our contract processor (Step ① in Figure 3.2) would need to be modified to automatically decompile and recompile unverified contracts to produce the required artifacts. Once these artifacts are obtained, the rest of HOUSTON’s pipeline is unchanged.

Private Transactions HOUSTON can prevent or detect abnormal transactions, whether before mining or after inclusion in a block. Clearly, prevention requires the ability to observe transactions in the mempool. However, under certain circumstances, this may not be feasible. For example, users might utilize private relayers (e.g., Flashbots [3]) or verifiers could insert transactions themselves before proposing a block. In such scenarios, a transaction will not be visible in the mempool, preventing HOUSTON from facilitating preemptive measures. This limitation could be mitigated if private relayers and/or verifiers adopt HOUSTON. Notably, the private industry is currently deploying systems similar to HOUSTON [52, 72, 75, 131], as a defense-in-depth strategy, demonstrating the value of such a system. These systems differentiate L2 services, which might censor attack transactions at the sequencer level, and support L1 blockchains in implementing protective measures when direct censorship is impractical.

Dynamic Adversary (Poisoning) Poisoning HOUSTON is possible in principle, but not without difficulty. Consider a contract where functions `A` and `B` can change the `openMarket` flag. While both functions should be protected by `onlyAdmin`, only `A` enforces it. An attacker observes that invoking `B` can close the market and impact the protocol, but the direct invocations of `B` would trigger an alert. A feasible poisoning strategy is to wait for an admin action executed via `A` that violates a **C1** invariant and, thus, generates an alert. If that alert is dismissed as an intentional admin action, HOUSTON would integrate the event into its notion of “normal” behavior. As a result, future transactions exhibiting the same behavior will not be flagged, allowing the attacker to call `B` to close the market without triggering an alert. This example shows that even when poisoning is feasible, HOUSTON still raises the difficulty of successful attacks: Adversaries must align their actions with rare administrative events *and* rely on the security team inadvertently dismissing the initial warning.

3.8 Related Work

Smart Contract Security The most effective way to protect DeFi protocols is to identify vulnerabilities during development, *before* deployment. To support this goal, the research community has explored several approaches: static analysis, formal verification, symbolic execution, and fuzzing. Static analysis is a technique used to scrutinize a program’s con-

tent and structure without executing it [16, 49, 56, 62, 173, 179]. Formal verification uses mathematical methods to prove adherence to certain security properties [53, 149, 160]. Symbolic execution explores a program’s behavior by executing code using symbolic inputs [15, 34, 66, 100, 118, 127, 141, 155, 171, 172]. Lastly, fuzzing involves repeatedly executing a target contract with many different generated inputs. [64, 71, 140, 170]. All these techniques have been used alone or in combination to identify both simple and complex bugs, such as integer overflows, reentrancy, excessive gas usage, and unsanitized calls. While identifying attacks before deployment is an important goal, it is often unattainable. Therefore, complementary approaches are required, such as detection (which is the focus of this work) and prevention.

Attack Detection Several prior works detect attacks by characterizing the patterns of actions performed by a user or a smart contract [74, 124, 161, 200]. These papers primarily focus on detecting Ponzi schemes, phishing, and other scams directed toward end users. DeFiRanger is a specialized tool to detect price manipulation attacks [184]. A handful of recent works attempt to tackle the topic of generalized anomaly detection for smart contracts. In this work, we discussed BLOCKGPT and APE [55, 133]. Other frameworks perform on-chain anomaly detection, but the alert conditions (i.e., assertions) must be manually annotated [22, 35, 153, 199]. Conversely, HOUSTON, does not need any transaction annotations to deliver off-chain anomaly detection. Several commercial anomaly detectors

have also been developed in the past few years [52, 72, 75, 77, 131]. These products generally require developers to manually define custom conditions that specify which behaviors should be permitted (or prohibited) during executions, or they rely on flagging transactions originating from suspicious sources such as sanctioned accounts or crypto mixers [169]. These conditions are evaluated either on-chain or off-chain, and any detected violation typically results in a warning or in the transaction being blocked. HOUSTON fundamentally differs from these approaches, as it eliminates the need for manually specified detection rules. Instead, HOUSTON’s models automatically infer the relevant signals for a given protocol. Furthermore, HOUSTON does not depend on external metadata such as transaction provenance, which is generally an unreliable signal prone to both false positives and false negatives.

Attack Prevention Various methods exist to prevent or mitigate an attack once it is recognized. Zhou et al. observe that about half of the DeFi protocols they surveyed include an emergency pause mechanism [205]. Other works develop automated syntheses of attack-stealing transactions, which are capable of performing the attack as “white hat” instead of the true attacker [133, 185, 202]. Notably, HOUSTON can act as the detection component that triggers such front-running tools.

Invariant Inference Daikon is a system that can infer likely invariants from program traces [43]. Liu et al. applied Daikon to smart contracts through their tool, InvCon,

which extracts simple invariants from ERC20 token contracts. However, InvCon has limitations, as it requires manual interpretation of invariant violations by a human operator. In contrast, HOUSTON provides a generic analysis to extract likely invariants from any smart contract, which can be continuously refined at runtime. This allows the invariants to adapt dynamically during live monitoring. Furthermore, HOUSTON automatically checks for violations, facilitating the timely detection of anomalous behavior. Chen et al. [23] utilized invariant templates to identify exploits related to known smart contract vulnerabilities, which requires manual annotation of template-relevant variables. In contrast, HOUSTON’s models are fully automated and can detect attack transactions as anomalies without requiring adaptation to specific attack types.

3.9 Conclusions

In this chapter, we introduce HOUSTON, a novel anomaly detection system for identifying attacks against Ethereum-based DeFi protocols. HOUSTON focuses on detecting deviations in protocol execution rather than recognizing known vulnerability patterns. HOUSTON distinguishes itself by (1) detecting anomalies through the analysis of the protocol interactions and likely invariants, and (2) adapting and evolving by ingesting new data from transactions. For evaluation, we used a comprehensive dataset of 115 attacks on DeFi protocols. The efficacy of HOUSTON, demonstrated through a comparative analysis

with state-of-the-art systems, shows its superior detection rate of 94.8% with a low false-positives rate of 0.16%. Experiments on live traffic confirm HOUSTON’s efficiency and its low false-positive rate in real-world deployments.

3.10 Appendix

3.10.A Storage Variables Decoder

The storage layout report emitted by the compiler provides a high-level overview of the storage structure. However, this report does not allow direct mapping of a given storage access (i.e., `SSTORE`) to the associated source code variables. For example, the storage layout may show that at slot id 0 there is a static array `my_struct[100]` with 100 *contiguous* elements – where `my_struct` is a simple data structure with few fields packed in a single storage slot. Given an `SSTORE` to slot `0x3`, to understand which variables are being overwritten, one would need to automatically recognize that slot id `0x3` still falls within the definition of the static array. To perform the translation from the compiler-

Table 3.1: Example of the `KECCAK256` look-up table collected during the transaction tracing (hashes are truncated for readability). To resolve an `SSTORE` that writes at slot id `0xd44ee5c9`, one can do a reverse-lookup in the table and extract the `BASE` in the pre-image until we identify a static slot. For example, here the look-up sequence would be: `0xd44ee5c9`→`0xbc36789e`→`0x0`. This shows the access pattern [`0x0`, `KECCAK256`, `KECCAK256`] that reveals access to an element of a nested mapping.

Preimage	Image
<code>admin.0x0</code>	<code>0xbc36789e</code>
<code>savings.0xbc36789e</code>	<code>0xd44ee5c9</code>

generated storage report to a programmatically usable data structure, we implemented an algorithm that recursively resolves every type definition within the storage layout and provides a “flat” view of the contract’s storage structure.

Complex Variable Types Complex variable types, such as mappings, and nested data structures are characterized by storage access patterns that use the `KECCAK256` hash function to derive the accessed slot id. These ids are calculated using either the accessed index (for a dynamic array) or the accessed key (for a dynamic mapping). As a result, since there are countless ways to access the same data structure, it is impractical to enumerate all of them in the storage layout report. Instead, we characterize accesses to dynamic slots by analyzing the sequence of operations that the EVM executes (during a transaction) to calculate the slot id. For example, consider the `KECCAK256` look-up table structure shown in Table 3.1, resulting from the look-up `users[“admin”][“savings”]`. In this example, `users` is a nested data structure defined as `map1[string:map2[string:uint256]]`. The look-up is performed with: `KECCAK256(KEY2.(KECCAK256(KEY1.BASE1))`, where `KEY1= “admin”`, `BASE1=0x0`, `KEY2=“savings”`, and `BASE2=0xbc36789e`. In this example, the elements nested within `map2` are uniquely identified by the access pattern `[0x0, KECCAK256, KECCAK256]`.

Given a slot id, the (flattened) contract’s storage layout, and the `KECCAK256` look-up table collected during the transaction tracing, the function SVD will: (1) if the slot id cor-

responds to a static slot, simply return the corresponding slot information from the storage layout; otherwise, (2) if the slot id is calculated using the KECCAK256 hash function (e.g., `0xd44ee5c9` in Table 3.1), iteratively perform reverse-lookups of the image within the KECCAK256 look-up table (Table 3.1) to identify the corresponding dynamic data structure and its base slot id. Once this is identified, the *SVD* procedure returns precise slot packing information using the storage layout.

3.10.B Dataset Overview

We observe a significant variation in the number of smart contracts in the protocols, the number of transactions, and the protocol’s lifetime at the moment of the attack. We compute the lifetime of a protocol from the first time that any of its contracts was deployed.

Table 3.2: Overview of DeFi Attacks Dataset.

	Transaction # per Protocol	Contract # per Protocol	Lifespan of Protocol (days)	Date of Exploitation
mean	74,662	131	344	2022-11-24
min	5	1	1	2020-01-10
25%	776	12	54	2022-01-14
50%	7,760	42	192	2023-03-13
75%	57,282	126	534	2023-11-27
max	1,079,908	1,378	2,133	2024-09-03

3.10.C Ablation Study

In this chapter, we argue that high-quality signals are critical for obtaining accurate detection results. To support this claim, this section presents the results of two ablation studies.

Protocol Interaction versus All Interactions First, we show how focusing on critical functions (§ 3.5.3) – rather than on *all* function calls – in a trace is crucial to lower false positives. To this end, we rerun our experiments with an *Interaction Model* that computes fingerprints based on all function calls. Figure 3.6 shows the results and demonstrates that the false positives drastically increases when using the full call trace. Specifically, without focusing on critical functions, *Interaction Model* produces 192 times more false positives (1,258,841) than the paper setup that facilitates the interaction simplification techniques. This highlights the importance of selective attention to relevant interactions for achieving accurate detection.

Fine- versus Coarse-Grained Storage Analysis The ability to identify the type and layout of storage variables (fine-grained storage analysis) is important for precise invariant generation, especially when protocols pack multiple variables in the same storage slot. For example, the value `0x40301020` might represent two different `uint16` variables – `0x4030` and `0x1020` – used in two different parts of the program. Interpreting the whole slot as a single `uint256` variable (coarse-grained storage analysis) would lead to imprecise

cise (or incomplete) invariants that negatively affect the decisions of the *Invariant Model*.

To assess the impact of the analysis granularity, we modify the function $\mathcal{SVD}$ (§ 3.5.1) to always report *one* single variable per slot of type `uint256`.

The *Invariant Model* detected four additional attacks with fine-grained storage analysis. We use the `sealfinance` incident to illustrate how coarse-grained analysis can lead to imprecise detection. The attack transaction violated the invariant in function `swap`: the argument `amountOut` must be less or equal to storage variable `reserve`. The coarse-grained analysis failed to identify this invariant because `reserve` was packed in the same slot with two other variables.

For false positive, the two configurations had similar performances, with fine-grained analysis yielding a slightly lower (0.7%) total number of false alarms. With more refined storage understanding, the precision of invariants improves. Meanwhile, the sheer number of variables and invariants also grows, which inevitably leads to an increase in false positives. These two opposing forces play out differently across protocols, and tend to balance out when considered across the entire dataset.

3.10.D Invariant Model Hyperparameter

The *Invariant Model* leverages two hyperparameters (§3.5.5) to tune the confidence of invariants. For an invariant to be considered valid, it has to be backed up by a certain

number of transactions ($N = 10$), and the contract has to be deployed for a minimum time span ($T = 12$ hours). We chose these values with common sense and the experience with a pilot study on `euler` and `inversefinance`.

After evaluating HOUSTON on the whole dataset, we investigated our choice of hyperparameters. Interestingly, we observed that there is a wide range of reasonable setups with little effect on the attack detection result (see Figure 3.7). Under our setup, the *Invariant Model* happened to miss two attacks (`btfinance`, `olympusdao`) as they both had violations against invariants inferred with 9 previous transactions while we required 10 or more observations for an invariant to be considered valid. Our *Interaction Model*, and thus HOUSTON, however, caught both attacks.

Higher N and T reduce the number of false positives, while the risk of missing attacks increases. An “overfit-to-dataset” hyperparameters setup ($N = 1$, $T = 48$) will have the same detection outcome as our more generic setup while yielding 11% fewer false alarms. Of course, we kept our initial setup and reported all results in the generic setting.

3.10.E Live Experiments Performance Statistics

HOUSTON performs the following procedures during operation: (1) `Trace`: Upon spotting a new transaction, HOUSTON collects the execution trace (using our custom plugin) from the Ethereum node. (2) `Process`: HOUSTON parses the raw execution trace,

Table 3.3: HOUSTON execution time (in second).

	Trace	Process	Detect	End-to-End
mean	0.003	0.180	0.028	0.211
median	0.003	0.133	0.013	0.148
99%	0.006	1.105	0.191	1.165
max	0.337	2.912	0.810	3.085

Table 3.4: Transaction load for HOUSTON during live experiments (number of transactions per second).

	Filter	Trace	Process / Detect
mean	24.9	12.5	0.1
median	24.2	12.1	0.0
99%	46.4	24.1	0.4
max	725.0	362.9	2.2

extracts relevant information, and turns it into the format needed for further analysis. (3)

Detect: The *Interaction Model* and the *Invariant Model* check for anomaly and raise warnings if necessary. HOUSTON then integrates new transaction data into its models.

The time statistics for these procedures, along with the end-to-end execution time (the cumulative time taken by steps (1)-(3)), are detailed in Table 3.3. We also provide statistics about the load of Ethereum mainnet traffic HOUSTON handled during the live experiments in Table 3.4.

3.10.F High TVL protocols

The boundaries of the protocols listed in Table 3.5 are approximated starting from the reference contract as discussed in §3.6.1. We picked as *reference contract* the one listed on *defillama.com* [37].

Chapter 3. Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

Table 3.5: Evaluation results on high TVL protocols with *no known* history of exploitation. *Duration* is measured in days.

	Protocol	Tot TXs	Duration	FP	(%)
1	ethena	589,317	123	12	(0.0%)
2	renzo	230,479	94	17	(0.0%)
3	etherfi	494,229	136	19	(0.0%)
4	ondofinance	496,387	821	24	(0.0%)
5	stakestone	417,500	342	63	(0.0%)
6	aave	2,558,803	1,406	120	(0.0%)
7	uniswap	2,868,003	1,412	132	(0.0%)
8	rocketpool	944,573	1,030	288	(0.0%)
9	stargatefinance	1,754,758	863	301	(0.0%)
10	everclear	41,491	480	42	(0.1%)
11	uncxnnetwork	246,268	1,381	133	(0.1%)
12	stakewise	99,950	1,278	140	(0.1%)
13	fraxfinance	179,405	665	244	(0.1%)
14	pendle	458,250	1,186	384	(0.1%)
15	tethergold	46,787	1,235	75	(0.2%)
16	aura	207,711	779	859	(0.4%)
17	convexfinance	1,845,533	1,166	7,396	(0.4%)
18	beefy	11,970	338	66	(0.6%)
19	hasnote	5,007	423	175	(3.5%)
	Total	13,496,421	-	10,488	(0.08%)

3.10.G Additional True Positive Examples

Polynetwork The attack was detected by a violation of a class **C2** invariant of the argument `curEpochPkBytes` in the function `putCurEpochConPubKeyBytes` callsite. This is in direct correlation with the attack vector: The attacker exploited a confused deputy vulnerability [66] to call `putCurEpochConPubKeyBytes` and modify the admin keys [150].

Audius The attack violated three **C2** and two **C3** invariants tied to governance settings, which were aggressively altered to enable malicious proposals. For example, the at-

tacker set `undelegateLockupDuration` to 1, breaking its historical parity with `removeDelegatorLockupDuration`.

Coverprotocol The attack was detected by the *Invariant Model* as a violation of a class **C2** invariant involving the variables `_totalSupply` and `_amount` when entering the function `mint()`. Historically, during minting, `_amount` was always less than `_totalSupply`. However, in this attack, over 40 quintillion COVER tokens were minted, causing the invariant to be violated.

Baconprotocol The attack was detected by the *Interaction Model*. Specifically, the sequence of actions `lend-lend-lend-redeem` was never observed before in the history of the protocol's execution. This strongly correlates with the root cause issue of the attack: a reentrancy bug in the `lend()` function [19].

BarleyFinance The attack was detected by both our models. The *Interaction Model* registered a very unique and long sequence of repeated actions `(flash-bond)n` which exactly match the attack technique: a *flash-loan-based reentrancy attack* [115]. On the other hand, the *Invariant Model* spotted a violation of a **C3** invariant: the value of `_totalSupply` was suddenly lower than the `_amount` arguments in the final call to the function `debond`.

3.10.H LLM prompts

The following are the system and user prompts we used to decide if the comparison between two variables in a smart contract is semantically meaningful or not.

Prompt 1: System Prompt

```
You are an expert smart contract developer and auditor.
Your language of expertise is Solidity/Vyper.
You will be given the code of a smart contract, followed by two
variables that appear in the code.
For instance, you might be given:
VAR1:  totalSupply
VAR2:  balance
Your task is to decide if it makes sense to compare the two
variables in the context of the smart contract.
For instance, if they are referring to two completely different
concepts, then the relationship is not meaningful.
*** IMPORTANT ***
1.  YOU MUST EMIT A REPORT IN THE FORMAT SPECIFIED BELOW.
2.  THE SUMMARY SHOULD EXPLAIN THE REASONING BEHIND THE
    DECISION.
3.  I WANT YOU TO BE CONSERVATIVE IN YOUR ASSESSMENT, IN OTHER
    WORDS, DISCARD RELATIONSHIPS THAT ARE NOT CLEARLY MEANINGFUL.
    IF YOU ARE NOT SURE, ANSWER YES.
Let's think step by step.
```

Chapter 3. Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

Prompt 2: User Prompt

```
You are given this smart contract code:
<contract>
TEMPLATED_CONTRACT_CODE
</contract>
Are these two variables semantically related and potentially
comparable in the context of the smart contract?
<variables>
TEMPLATED_RELATION
</variables>
After completing the analysis, you MUST output a report in the
specified format.
```
<report>
<verdict>
</verdict>
<summary>
...
</summary>
</report>
```
```

3.10.I HOUSTON Results Overview

Table 3.6: HOUSTON historical evaluation results. ✓ represents true positive, ✗ represents false negative. Column *Tot TXs* is the total amount of transactions available for the protocol. Column *Lifespan* represents the number of days the protocol had been active prior to the attack. Column *FP* shows the total amount (and percentage) of false positive transactions in the *Tot TXs*. Column *FP/d* shows the average daily false positive alerts for a protocol. *Inv. FP* and *Int. FP* are the false positive counts of the *Invariant* and *Interaction* Models respectively; *Inv.* and *Int.* are their decisions on the attack transaction. Symbol – means the protocol was not in the baseline’s dataset. For BlockGPT, ✓_{1/2/3} indicates 1st/2nd/3rd most abnormal. We use ★ for protocols monitored live. Abbreviation in table: B=BlockGPT [55]; A=APE [133]; TS=TxSpector [199]; DR=DeFiRanger [184]; Lg=longest-tx baseline.

	Protocol	Tot TXs	Lifespan	FP (%)	FP/d	Inv. FP	Int. FP	Inv.	Int.	HOUSTON	B	A	TS	DR	Lg
1	polynetwork ★	57,140	363	3 (0.0%)	0.0	0	3	✓	✗	✓	✗	✗	✗	✗	✗
2	grok	66,790	6	7 (0.0%)	1.2	0	7	✗	✓	✓	-	-	✓	✗	✗
3	teamfinance ★	30,619	456	15 (0.0%)	0.0	0	15	✗	✓	✓	-	-	✗	✗	✓
4	mcc	59,801	536	26 (0.0%)	0.0	12	14	✓	✓	✓	-	-	✓	✓	✓
5	loopring ★	248,781	1142	36 (0.0%)	0.0	13	26	✓	✗	✓	-	-	✗	✓	✗

Continued on next page

Chapter 3. Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

Table 3.6 – continued from previous page															
	Protocol	Tot TXs	Lifespan	FP (%)	FP/d	Inv. FP	Int. FP	Inv.	Int.	HOUSTON	B	A	TS	DR	Lg
6	umbrellanetwork *	139,042	405	42 (0.0%)	0.1	27	15	✓	✗	✓	-	✗	✗	✗	✗
7	shanshu *	173,121	91	42 (0.0%)	0.5	28	15	✗	✓	✓	-	-	✗	✓	✗
8	audius	268,109	640	77 (0.0%)	0.1	45	33	✓	✓	✓	-	✗	✗	✗	✗
9	kashi *	753,725	531	128 (0.0%)	0.2	35	97	✓	✓	✓	-	-	✗	✓	✗
10	sandboxland	1,079,908	833	231 (0.0%)	0.3	105	134	✗	✓	✓	-	✗	✗	✗	✗
11	game	4,276	1	6 (0.1%)	6.0	0	6	✗	✓	✓	-	-	✓	✗	✓
12	dexible	22,609	1074	24 (0.1%)	0.0	5	19	✗	✓	✓	-	-	✗	✗	✗
13	btc20	55,054	66	30 (0.1%)	0.5	12	21	✗	✓	✓	-	-	✗	✓	✓
14	compoundfinance	40,993	942	37 (0.1%)	0.0	20	20	✓	✓	✓	-	-	✗	✓	✗
15	kyberswap	59,986	767	42 (0.1%)	0.1	20	22	✗	✓	✓	-	-	✗	✗	✗
16	nfttrader	42,115	998	45 (0.1%)	0.0	23	22	✓	✓	✓	-	-	✗	✗	✗
17	balancer	61,931	31	57 (0.1%)	1.8	16	41	✓	✓	✓	✗	-	✓	✓	✓
18	coverprotocol	105,424	64	61 (0.1%)	1.0	18	45	✓	✗	✓	✗	-	✗	✗	✗
19	sealfinance	65,593	53	96 (0.1%)	1.8	73	25	✓	✗	✓	-	-	✓	✓	✗
20	thorchain *	213,250	191	120 (0.1%)	0.6	93	30	✓	✗	✓	✗	-	✗	✗	✗
21	harvestfinance *	217,992	56	151 (0.1%)	2.7	137	16	✗	✓	✓	✗	-	✓	✓	✓
22	indexedfinance *	211,104	317	214 (0.1%)	0.7	89	132	✓	✓	✓	✓ ₃	✗	✓	✗	✓
23	mimspell	390,419	980	376 (0.1%)	0.4	115	280	✓	✓	✓	-	-	✗	✗	✗
24	wiselending	382,780	2133	415 (0.1%)	0.2	173	259	✗	✓	✓	-	-	✗	✗	✗
25	valuedefi *	655,019	90	489 (0.1%)	5.4	384	110	✓	✓	✓	✓ ₁	-	✗	✓	✗
26	bzxprotocol	365,787	1239	515 (0.1%)	0.4	338	199	✗	✓	✓	-	-	✗	✓	✗
27	visorfinance	1,745	110	3 (0.2%)	0.0	0	3	✗	✓	✓	✓ ₁	✗	✗	✗	✓
28	tinu	3,668	586	6 (0.2%)	0.0	1	5	✓	✓	✓	-	-	✗	✓	✗
29	shoco	7,760	594	16 (0.2%)	0.0	12	6	✓	✓	✓	-	-	✓	✓	✗
30	ktaf	37,104	1291	89 (0.2%)	0.1	40	50	✓	✓	✓	-	-	✗	✓	✓
31	conicfinance *	47,208	477	111 (0.2%)	0.2	53	67	✓	✓	✓	-	-	✗	✗	✓
32	klondikefinance	60,204	231	129 (0.2%)	0.6	98	38	✓	✓	✓	✓ ₁	✓	✗	✗	✗
33	alphafinance *	57,424	130	135 (0.2%)	1.0	98	48	✓	✓	✓	✗	-	✗	✗	✗
34	floorprotocol	106,524	65	226 (0.2%)	3.5	191	35	✗	✓	✓	-	-	✗	✗	✗
35	inversefinance *	139,687	478	247 (0.2%)	0.5	136	123	✗	✓	✓	✓ ₂	✗	✗	✓	✗
36	creamfinance *	587,106	393	958 (0.2%)	2.4	717	275	✗	✓	✓	✗	✗	✗	✗	✗
37	penpiexyzio	844,841	688	2,010 (0.2%)	2.9	1,333	713	✗	✓	✓	-	-	✗	✓	✗
38	deeznutz	860	1	3 (0.3%)	3.0	0	3	✗	✓	✓	-	-	✗	✗	✓
39	btfinance	11,483	69	38 (0.3%)	0.6	20	18	✗	✓	✓	✓ ₁	-	✗	✓	✓
40	xstableprotocol	15,735	563	48 (0.3%)	0.1	26	24	✓	✓	✓	-	-	✗	✗	✓
41	revestfinance *	22,438	187	57 (0.3%)	0.3	30	31	✓	✓	✓	✓ ₁	✗	✓	✗	✗
42	floordao	25,605	1081	78 (0.3%)	0.1	27	56	✓	✓	✓	-	-	✓	✗	✓
43	saddlefinance	47,308	466	132 (0.3%)	0.3	95	43	✗	✓	✓	✗	-	✓	✗	✗
44	uniclynft	56,558	900	149 (0.3%)	0.2	77	80	✗	✓	✓	-	-	✓	✗	✗
45	hypr	1,137	29	5 (0.4%)	0.2	0	5	✓	✓	✓	-	-	✗	✗	✓
46	veth	7,729	52	33 (0.4%)	0.6	21	13	✓	✗	✓	✓ ₃	-	✓	✗	✗
47	raftfi	31,214	750	128 (0.4%)	0.2	69	70	✗	✓	✓	-	-	✗	✓	✓
48	euler *	101,868	469	421 (0.4%)	0.9	257	182	✓	✓	✓	-	-	✗	✗	✗
49	0x0dex	2,590	106	12 (0.5%)	0.1	8	4	✓	✓	✓	-	-	✗	✗	✓
50	sodafinance	7,566	2	40 (0.5%)	20.0	28	16	✗	✓	✓	✗	-	✗	✗	✗

Continued on next page

Chapter 3. Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

Table 3.6 – continued from previous page

	Protocol	Tot TXs	Lifespan	FP (%)	FP/d	Inv. FP	Int. FP	Inv.	Int.	HOUSTON	B	A	TS	DR	Lg
51	dodo ★	19,218	46	94 (0.5%)	2.0	42	55	✓	✓	✓	✓ ₁	-	✗	✗	✗
52	picklefinance	33,367	54	182 (0.5%)	3.4	139	56	✓	✓	✓	✗	-	✗	✗	✗
53	qnttoken	1,877	541	12 (0.6%)	0.0	5	8	✗	✓	✓	✓	-	-	✗	✓
54	thenftv2	14,194	923	88 (0.6%)	0.1	50	45	✗	✓	✓	✓	-	-	✗	✗
55	prismafi	134,389	248	871 (0.6%)	3.5	144	743	✗	✓	✓	✓	-	-	✗	✗
56	bbt	140	54	1 (0.7%)	0.0	0	1	✓	✓	✓	✓	-	-	✓	✗
57	jay	1,000	747	8 (0.8%)	0.0	1	7	✓	✓	✓	✓	-	-	✗	✗
58	xtoken	9,206	116	73 (0.8%)	0.6	33	42	✓	✓	✓	✗	✗	✓	✗	✓
59	zenterest	85,332	1325	716 (0.8%)	0.5	651	82	✗	✓	✓	✓	-	-	✗	✗
60	pineprotocol	10,749	684	92 (0.9%)	0.1	25	70	✗	✓	✓	✓	-	-	✗	✗
61	n00dtoken	2,016	49	22 (1.1%)	0.4	12	10	✓	✓	✓	✓	-	-	✓	✗
62	popsiclefinance	3,909	40	43 (1.1%)	1.1	38	5	✓	✓	✓	✓ ₁	✓	✗	✗	✓
63	blueberryprotocol	1,520	32	19 (1.2%)	0.6	9	11	✓	✓	✓	✓	-	-	✓	✗
64	dfxfinance	3,243	344	39 (1.2%)	0.1	10	29	✗	✓	✓	✓	-	-	✓	✗
65	olympusdao ★	3,557	254	44 (1.2%)	0.2	5	40	✗	✓	✓	✓	-	-	✗	✗
66	convergence	11,960	184	147 (1.2%)	0.8	46	105	✓	✗	✓	✓	-	-	✗	✗
67	deusdao	1,339	209	18 (1.3%)	0.1	9	12	✓	✗	✓	✓	-	-	✗	✗
68	barleyfinance	2,294	3	29 (1.3%)	9.7	10	19	✓	✓	✓	✓	-	-	✗	✓
69	affinedefi	4,532	523	60 (1.3%)	0.1	11	51	✓	✓	✓	✓	-	-	✗	✗
70	paraspacenft	40,264	106	574 (1.4%)	5.4	72	518	✓	✓	✓	✓	-	-	✗	✗
71	luckytigernft	579	28	10 (1.7%)	0.4	4	6	✗	✓	✓	✓	-	-	✗	✓
72	fulcrum ★	20,309	421	349 (1.7%)	0.8	270	81	✗	✓	✓	✓	✗	-	✗	✗
73	filxdn404	339	107	6 (1.8%)	0.1	0	6	✓	✓	✓	✓	-	-	✗	✓
74	sturdyfinance	8,479	384	159 (1.9%)	0.4	75	99	✓	✓	✓	✓	-	-	✗	✓
75	auctus	4,617	675	98 (2.1%)	0.1	58	44	✓	✓	✓	✓	-	✓	✗	✓
76	punkprotocol	322	113	7 (2.2%)	0.1	2	5	✓	✓	✓	✓ ₁	✗	✗	✗	✓
77	gain	1,574	581	34 (2.2%)	0.1	12	28	✗	✓	✓	✓	-	-	✓	✓
78	zunamiprotocol	5,850	522	127 (2.2%)	0.2	44	91	✓	✗	✓	✓	-	-	✗	✓
79	baconprotocol	1,148	220	27 (2.4%)	0.1	5	22	✗	✓	✓	✓	-	✗	✗	✗
80	warpfinance	475	2	12 (2.5%)	6.0	4	8	✗	✓	✓	✓	✓ ₁	-	✗	✓
81	rubic	725	122	18 (2.5%)	0.1	9	9	✓	✓	✓	✓	-	-	✗	✓
82	monoxfinance	1,066	46	27 (2.5%)	0.6	14	13	✓	✓	✓	✓	-	✗	✗	✓
83	xcarnival	1,149	21	30 (2.6%)	1.4	17	15	✗	✓	✓	✓	-	✗	✗	✓
84	paraswap	686	7	19 (2.8%)	2.7	6	13	✗	✓	✓	✓	-	-	✗	✗
85	wildcredit	2,913	33	83 (2.8%)	2.5	37	50	✓	✓	✓	✓ ₁	-	✗	✗	✗
86	cheesebank	1,800	41	52 (2.9%)	1.3	37	16	✗	✓	✓	✓ ₁	-	✗	✓	✓
87	juice	93	1	3 (3.2%)	3.0	0	3	✗	✓	✓	✓	-	-	✗	✗
88	astridprotocol	371	22	15 (4.0%)	0.7	8	9	✗	✓	✓	✓	-	-	✓	✓
89	uwerx	145	161	6 (4.1%)	0.0	4	2	✓	✓	✓	✓	-	-	✗	✗
90	templedao	545	133	25 (4.6%)	0.2	5	23	✓	✓	✓	✓	-	-	✗	✗
91	particetrade	303	161	16 (5.3%)	0.1	2	15	✓	✓	✓	✓	-	-	✗	✗
92	peapodsfinance	712	641	38 (5.3%)	0.1	8	34	✗	✓	✓	✓	-	-	✗	✗
93	ruggedart	116	10	7 (6.0%)	0.7	0	7	✗	✓	✓	✓	-	-	✓	✗
94	gooddollar	778	752	48 (6.2%)	0.1	16	35	✗	✓	✓	✓	-	-	✓	✓
95	hopelend	538	73	34 (6.3%)	0.5	11	25	✓	✓	✓	✓	-	-	✗	✓

Continued on next page

Chapter 3. Houston: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

Table 3.6 – continued from previous page															
	Protocol	Tot TXs	Lifespan	FP (%)	FP/d	Inv. FP	Int. FP	Inv.	Int.	HOUSTON	B	A	TS	DR	Lg
96	omninf	255	26	19 (7.5%)	0.7	3	16	✓	✓	✓	-	✗	✗	✗	✓
97	levusdc	118	213	9 (7.6%)	0.0	3	7	✗	✓	✓	-	-	✓	✗	✗
98	dappsocial	109	59	9 (8.3%)	0.2	0	9	✗	✓	✓	-	-	✗	✗	✓
99	vinu	22	1	2 (9.1%)	2.0	0	2	✗	✓	✓	-	-	✗	✓	✓
100	pawnfi	775	55	77 (9.9%)	1.4	40	40	✓	✓	✓	-	-	✗	✗	✓
101	mahalend	223	334	23 (10.3%)	0.1	8	17	✓	✓	✓	-	-	✗	✓	✓
102	earningfram	635	306	71 (11.2%)	0.2	23	50	✓	✓	✓	-	-	✗	✗	✓
103	shata	556	140	64 (11.5%)	0.5	21	45	✗	✓	✓	-	-	✗	✗	✗
104	cowswap	298	206	35 (11.7%)	0.2	0	35	✗	✓	✓	-	-	✗	✗	✗
105	baocommunity	89	134	12 (13.5%)	0.1	1	12	✗	✓	✓	-	-	✗	✓	✓
106	onyxprotocol	5	6	1 (20.0%)	0.2	0	1	✗	✓	✓	-	-	✓	✗	✓
107	pandora	9	1	2 (22.2%)	2.0	0	2	✗	✓	✓	-	-	✗	✗	✓
108	yodlrouter	45	54	10 (22.2%)	0.2	4	6	✗	✓	✓	-	-	✗	✗	✗
109	burntbubba	56	985	15 (26.8%)	0.0	0	15	✗	✓	✓	-	-	✗	✓	✗
110	minerercx	20,844	3	15 (0.1%)	5.0	3	12	✗	✗	✗	-	-	✗	✗	✓
111	nomad	63,880	203	40 (0.1%)	0.2	24	21	✗	✗	✗	-	-	✗	✗	✗
112	azukidao	3,182	1	6 (0.2%)	6.0	4	2	✗	✗	✗	-	-	✗	✗	✗
113	snood	12,042	347	42 (0.3%)	0.1	30	12	✗	✗	✗	-	✗	✗	✗	✗
114	88mph ★	14,257	509	198 (1.4%)	0.4	108	92	✗	✗	✗	✓ ₁	-	✗	✗	✗
115	bzx	32,553	456	440 (1.4%)	1.0	352	90	✗	✗	✗	✗	-	✓	✓	✗
	TOTAL	8,586,421		13,827 (0.16%)		7,702	6,568								

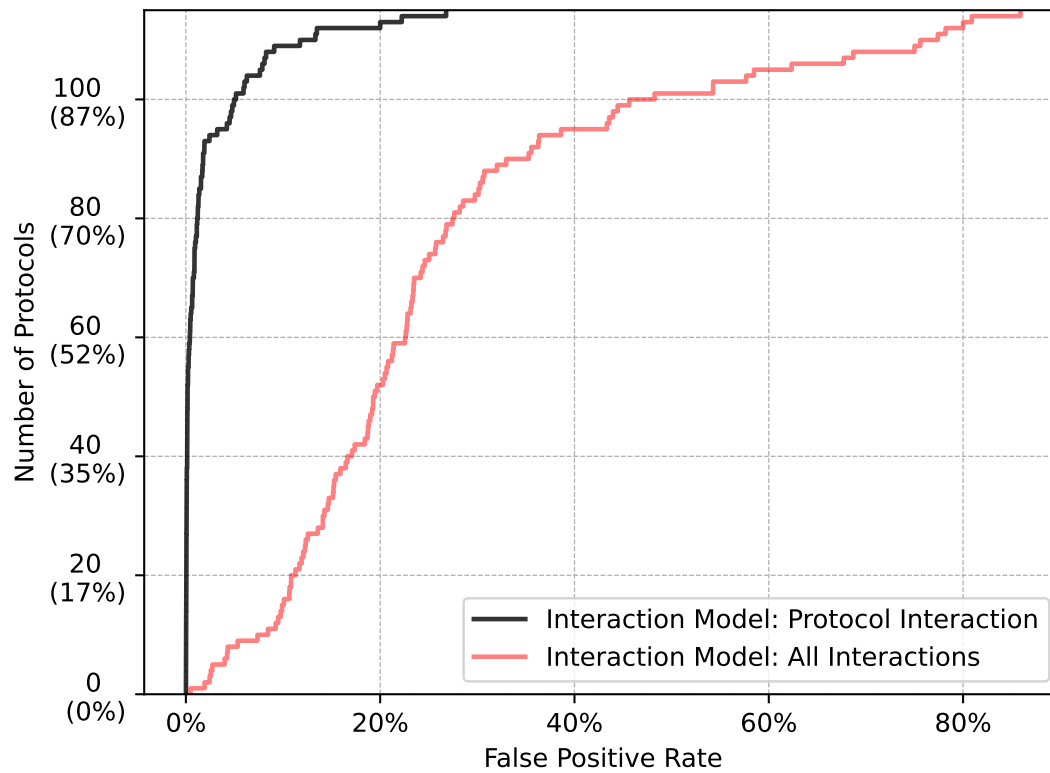


Figure 3.6: Cumulative distribution of false positive rates of the *Interaction Model* with two different configurations.

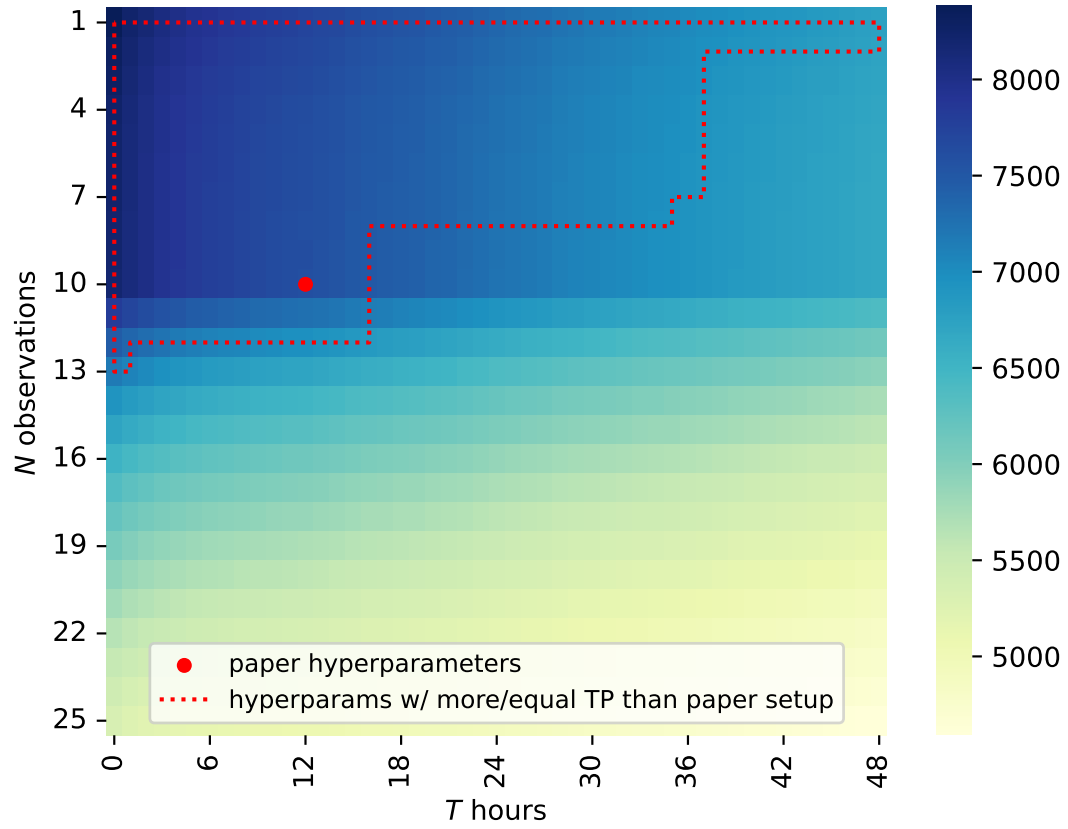


Figure 3.7: Total number of false positives of the *Invariant Model* across all protocols with different hyperparameter setups. The paper setup does not overfit to the dataset.

Chapter 4

Dissonance: Detecting Behavioral Regressions in Smart Contract Upgrades

4.1 Introduction

Blockchains such as Ethereum [183] and BNB Chain [13] have grown into a major financial infrastructure, hosting decentralized exchanges, lending markets, stablecoins, and governance systems that operate without central intermediaries. At the heart of this ecosystem are *smart contracts*: programs deployed on the blockchain that hold assets, enforce rules, and coordinate value transfers. The decentralized finance (DeFi) sector [181] now holds over \$100 billion in total value locked in smart contracts [37], processing transactions that range from individual swaps to multi-million-dollar institutional flows.

A smart contract is a program deployed on the blockchain and executed by the Ethereum Virtual Machine (EVM) [183]. Each contract maintains its own persistent storage, can hold and transfer the chain's native currency or tokens, and can invoke functions on other deployed contracts.

A defining property of contracts is that, once deployed, their bytecode is immutable and cannot be modified in place. While this is desirable for trust and reproducibility, it conflicts with the practical need to fix bugs, add features, or respond to changing requirements. Developers therefore rely on indirection: instead of binding to a single immutable contract, they deploy a *proxy* contract that holds assets and state at a stable address while delegating execution logic to a separate implementation contract that can be replaced [14, 147]. Upgrading a contract then consists of deploying a new implementation contract and updating the proxy's reference to point to the new implementation. From the outside, the contract's address remains constant, and its persistent state carries over seamlessly.

In practice, however, upgrades are far less orderly than the deployments they supplant. They are often executed under tighter timelines than the original audit cycle, sometimes mediated through governance processes with uneven levels of expert scrutiny, and applied atop complex, long-lived state and integrations. This combination creates a fragile moment in a contract's lifecycle.

A growing number of high-profile incidents demonstrate that contract upgrades can silently introduce vulnerabilities. In the Pike Finance incident [10] (April 2024, \$1.6M lost), insertion of a pausable module shifted the storage layout, causing the initialization flag to be overwritten with a zeroed byte and allowing anyone to reinitialize the contract and overwrite administrative parameters. In the Compound incident [139] (September 2021, ~\$160M lost), a community proposal restructured COMP reward distribution and rewrote the conditional gating retroactive reward attribution. The new conditional used $>$ where $\geq$ was required. For users whose per-market supplier index had been silently corrupted to zero by a latent bug in the prior version, the new logic computed reward deltas of $1e36$ times the user's cToken balance, allowing many users to claim massively inflated rewards. In the Nomad incident [30] (August 2022, ~\$190M lost), an upgrade to the bridge's message-verification logic began accepting the zero hash as a valid trusted root, and a value written into that root mapping during an earlier initialization made every unproven message pass verification, allowing anyone to copy and replay forged withdrawals.

What unites these incidents is not a shared vulnerability class but a shared property of the upgrade transition itself. An upgrade is rarely a clean substitution of one implementation for another: it composes new logic with pre-existing state, assumptions, and invariants that were established under a different program. As a result, subtle mismatches arise along several dimensions. Storage may be reinterpreted under a new layout (as in

Pike Finance), latent inconsistencies in persisted state may become exploitable when new logic operates over them (as in Compound), and values that were previously benign may acquire new meaning under revised validation rules (as in Nomad).

Reviewing an upgrade therefore requires reasoning about this composition explicitly: making the behaviors of the old and new implementations concrete on their shared state, so that divergences become visible rather than implicit. This is precisely the form of analysis that existing tooling does not provide. Prior work splits along two lines. One line reasons about upgrades structurally, without analyzing execution behavior. Tools such as USCHunt [14], CRUSH [141], and Slither [49] analyze proxy patterns to detect storage-layout clashes, selector collisions, and related anti-patterns. While effective for these classes of bugs, they are limited to structural inconsistencies. The other line reasons about behavior, but only within a single version. Static analyzers and symbolic execution engines [34, 53, 100, 173] detect vulnerability patterns in one implementation at a time, and dynamic invariant detectors [23, 96] infer properties from a single version’s execution traces. Neither line captures the interaction between versions: structural approaches do not analyze execution behavior, and behavioral approaches do not compare across versions.

In this chapter, we present DISSONANCE, a differential behavioral analysis system for smart contract upgrades. Rather than analyzing each version in isolation, DISSONANCE treats an upgrade as a composition of two implementations over a shared state and makes

their behaviors directly comparable. Given the old and new implementations behind the same proxy, DISSONANCE analyzes them under identical conditions and surfaces divergences that arise from the upgrade: missing access-control checks, weakened guard conditions, or changes in how persistent state is read and written.

To do so, DISSONANCE routes functions based on their relationship across versions. For *modified functions*, whose bytecode differs, DISSONANCE performs differential symbolic execution, collecting path conditions, storage effects, and external calls from both versions and comparing their behaviors across explored execution paths. For *newly introduced functions*, which have no prior counterpart, DISSONANCE applies a lightweight static analysis that compares their storage footprint and access-control structure against related functionality in the previous implementation.

DISSONANCE operates directly on EVM bytecode and requires no source code. It lifts bytecode to a three-address representation using the Gigahorse decompiler [61], recovers function interfaces from decompilation artifacts, and synthesizes ABI-aware calldata templates to control symbolic path explosion.

We evaluate DISSONANCE on a ground-truth dataset of 12 known upgrade-related incidents and a discovery dataset of 1,713 upgrade pairs collected from Ethereum. DISSONANCE detects 10 of the 12 known incidents. On the discovery dataset, manual triage of

DISSONANCE's reports uncovers 2 previously unreported vulnerable upgrades and provides a detailed characterization of its false positives across different upgrade patterns.

In summary, this chapter makes the following contributions:

- We introduce *differential behavioral analysis* for smart contract upgrades, a technique that compares two implementation versions over shared state to surface security-relevant divergences.
- We design a hybrid analysis that combines differential symbolic execution for modified functions with static cross-function analysis for newly introduced functions.
- We implement our approach in DISSONANCE and evaluate it on 1,713 real-world upgrade pairs from Ethereum, detecting 10 of 12 known incidents and surfacing 2 previously unreported vulnerabilities.

4.2 Background

4.2.1 Ethereum Smart Contracts

Ethereum [183] is a decentralized blockchain platform that supports the execution of *smart contracts*: programs that run on the Ethereum Virtual Machine (EVM). Smart contracts are typically written in high-level languages such as Solidity [166] or Vyper [167],

compiled to EVM bytecode, and deployed to the blockchain at a unique address. Once deployed, a contract's bytecode is immutable: it cannot be modified or patched.

The EVM is a stack-based virtual machine with a 256-bit word size. Each contract has access to a persistent key-value store called *storage*, where both keys and values are 256-bit words. Storage is the primary mechanism for maintaining state across transactions (e.g., token balances, access control variables, configuration parameters). Contracts modify their storage through `SSTORE` instructions and read from it via `SLOAD`. Contracts can also invoke other contracts through instructions like `CALL` and `DELEGATECALL`, enabling composability across the ecosystem.

Each publicly accessible function in a contract is identified by a 4-byte *function selector*, derived from the first four bytes of the Keccak-256 hash of the function's signature [156]. When a transaction targets a contract, the EVM dispatches execution to the function matching the selector in the transaction's calldata.

4.2.2 Upgradeable Smart Contracts and the Proxy Pattern

While the immutability of deployed bytecode provides strong integrity guarantees, it also prevents developers from fixing bugs or adding features after deployment. To work around this limitation, the community has adopted the *proxy pattern* [14, 147], which separates a contract's state from its logic.

In this pattern, users interact with a *proxy contract* that holds all persistent storage but contains minimal logic. When the proxy receives a transaction, it forwards the call to a separate *implementation contract* using the EVM's `DELEGATECALL` instruction. `DELEGATECALL` executes the implementation's code in the context of the proxy: all storage reads and writes operate on the proxy's storage, and `msg.sender` remains the original caller. The address of the current implementation is stored in a designated storage slot of the proxy.

To upgrade the contract, an authorized party (typically the contract owner or a governance mechanism) updates the implementation address in the proxy's storage to point to a new contract. From that point forward, all calls are forwarded to the new implementation, which operates on the same storage as its predecessor. This mechanism enables iterative development but introduces a significant attack surface: a malicious or buggy upgrade can silently alter the contract's behavior (e.g., weaken access controls, introduce unauthorized fund transfers, or corrupt storage invariants) without changing the contract's address or requiring users to take any action.

4.2.3 Symbolic Execution

Symbolic execution [100] is a program analysis technique that explores execution paths by replacing concrete inputs with symbolic variables and reasoning about the con-

straints under which each path is taken. Rather than executing a program with a single set of concrete inputs, a symbolic execution engine maintains a *symbolic state* consisting of symbolic expressions for registers and memory, along with *path constraints* that encode the conditions required to reach the current program point. At each branch, the engine queries a constraint solver (e.g., Z3 or Yices) to determine the feasibility of each path and forks the state accordingly.

In the context of EVM bytecode analysis, symbolic execution allows reasoning about all possible inputs to a contract function simultaneously. For example, when analyzing an access control check such as `require(msg.sender == owner)`, the symbolic engine tracks that execution beyond the check is only feasible when the caller equals the stored owner address. By capturing the path constraints and storage operations along each feasible path, symbolic execution produces a path-sensitive characterization of a function’s behavior: which storage slots it writes, under what conditions, and with what values.

DISSONANCE leverages symbolic execution to extract summaries from both the old and new implementations of an upgraded contract. By comparing these summaries, it identifies differences in storage effects, external calls, and path conditions that may indicate security-relevant regressions.

4.3 Approach

We frame upgrade security as a behavioral differencing problem: given two contract implementations sharing the same proxy storage, identify operations whose observable effects (storage writes, external calls, or execution conditions) differ between versions in security-relevant ways.

Figure 4.1 shows the analysis pipeline. The input is a pair of implementation contracts (old and new) deployed behind the same proxy, together with the block number at which the upgrade occurred. The output is a set of *findings*, each identifying a specific behavioral change and the supporting evidence (e.g., the affected storage slot, call target, or constraint expression).

The analysis proceeds in four stages. First, the bytecode of both implementations is decompiled to a three-address code (TAC) intermediate representation [61], from which public function selectors are extracted and classified as *modified*, *newly added*, or *removed* (§4.3.1). Second, modified functions undergo symbolic execution on both versions, producing behavioral profiles that capture storage writes, external calls, and path constraints (§4.3.2). Third, these profiles are structurally differenced (§4.3.3–4.3.4) and passed to a suite of detectors that flag security-relevant regressions (§4.3.5). In parallel, newly added functions are analyzed via a lightweight static pass that compares their storage footprint and access-control structure against related functions in the prior version (§4.3.6).

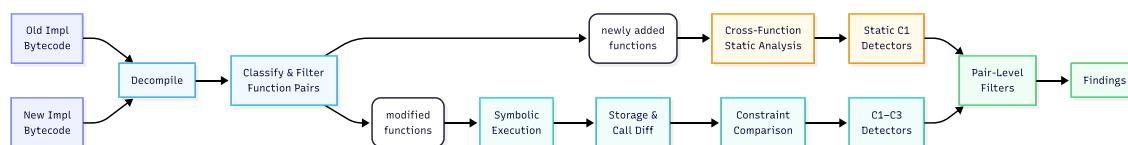


Figure 4.1: Overview of the DISSONANCE analysis pipeline. Bytecode for both implementations is decompiled to TAC and each public function is classified as modified, newly added, or removed. Modified functions are symbolically executed on both versions; the resulting behavioral profiles (storage writes, external calls, path constraints) are structurally differenced and fed to detectors spanning three classes of behavioral divergence (C1 new reachability, C2 weakened conditions, C3 value divergence). Newly added functions are analyzed statically for broken guard lifecycles and unguarded admin writes (C1). Pair-level filters suppress systematic false positives before producing the final findings.

The approach operates entirely on bytecode; source code is not required. Function signatures and parameter types are recovered from decompilation artifacts, making the methodology applicable to unverified contracts for which no source is available. When verified source is available for both implementations, it serves as an additional pre-simulation filter (§4.3.1).

4.3.1 Function Classification and Filtering

The first step is to identify which functions were modified between versions. Given the TAC representations of both implementations, we enumerate the public functions of each version by their 4-byte selectors and classify each selector as *modified*, *newly added*, or *removed*.

Determining whether a function’s code has been modified requires care, because the decompiler assigns basic block identifiers based on bytecode offsets, which shift when

the compiler or optimizer produces a different bytecode layout. A naive comparison of block-level code would report spurious differences even when the logic is identical. We address this through *composite fingerprints* that capture the function’s entire reachable callgraph in a layout-independent way. A single function’s fingerprint is the sequence of opcodes across its basic blocks, ordered by block content rather than block identifier. The composite fingerprint extends this to the reachable callgraph: the set of fingerprints of all internal functions reachable via private calls. Two selectors with identical composite fingerprints are classified as unchanged, regardless of differences in block identifiers or internal function numbering.

Not all modified functions require symbolic execution. We apply four conservative pre-simulation filters:

1. **State-changing filter.** Functions whose reachable callgraph contains no state-modifying opcode (SSTORE, CALL, DELEGATECALL, CREATE, or CREATE2) are skipped. Any code change in such functions is invisible at the behavioral level.
2. **State-diff filter.** Functions where the code change is confined to non-state-modifying helpers are skipped. The composite fingerprint is recomputed over state-changing callees only; if it matches, the behavioral profile is guaranteed identical.
3. **Opcode-bag filter.** Functions whose reachable-callgraph opcode multiset (counting each opcode name per basic block) is identical between old and new are skipped. This

targets compile-setting drift where the compiler rearranges basic blocks or changes jump targets without altering the set of executed instructions.

4. **Source-identity filter.** When verified source is available for both implementations, functions whose source bodies are character-identical after whitespace normalization are skipped. This catches optimizer-toggle drift, where identical source compiles to different bytecode.

Filters 1–3 operate purely on bytecode. Filter 4 is an opportunistic optimization applicable when verified source exists. All filters apply exclusively to the symbolic execution track. Newly added functions always proceed to the static analysis track.

4.3.2 Behavioral Profile Extraction

For each modified function that passes the pre-simulation filters, both the old and new implementations are symbolically executed to produce a *behavioral profile*: a structured record of the function’s observable effects and the conditions under which they occur.

Calldata Generation Each function’s ABI signature specifies its parameter types and encoding [156]. We parse this signature into a tree of typed parameters and generate a set of *calldata templates*: concrete byte-level layouts that fix the structural aspects of the encoding (array lengths, tuple nesting depths) while leaving parameter values symbolic. Templates cover boundary cases for numeric types, empty and non-empty dynamic arrays,

boolean combinations, and nested tuple structures. Each template is executed independently, and the union of results constitutes the function’s behavioral profile. This decomposition manages path explosion by confining the symbolic executor to value-dependent branches within a fixed structural layout.

Execution Model DISSONANCE performs symbolic execution using a modified version of Greed [142], a symbolic execution engine for EVM smart contracts that operates on the TAC intermediate representation produced by the Gigahorse decompiler [61]. Since proxy-pattern contracts execute implementation code via `DELEGATECALL` in the proxy’s storage context (§4.2.2), the symbolic store is initialized with the proxy’s actual on-chain storage values at the upgrade block. This *partial concrete storage* grounds the analysis in real contract state, resolving storage-dependent branches against actual values while keeping the function’s own writes symbolic. Loop iterations are bounded at a configurable limit (five by default) to ensure termination.

Profile Contents For each non-reverting execution path, three classes of observations are recorded. *Storage writes*: each `SSTORE` instruction with its symbolic key expression, symbolic value expression, and the path constraint. *External calls*: each `CALL`-family instruction with its target address expression, call value, and argument expressions. *Path constraints*: the set of branch conditions that the solver verified as satisfiable to reach this

program point. Each terminal state receives a unique path identifier, preserving path-level granularity through serialization and caching.

4.3.3 Storage and Call Comparison

Given the behavioral profiles of a function’s old and new implementations, the next step is to compute a structured comparison that classifies every storage write and external call into one of four categories.

Categories Let W_{old} and W_{new} denote the sets of storage-write records from the old and new implementations, where each record is a triple (k, v, φ) consisting of a key expression k , a value expression v , and a path constraint φ . The comparison partitions these records as follows:

- **Matched.** The key expression is structurally equivalent across versions, and so is the value. The operation is behaviorally unchanged.
- **Changed.** The key matches, but the value expression differs.
- **Only-old.** The key appears in W_{old} but has no structural counterpart in W_{new} . A previously existing write was removed.
- **Only-new.** The key appears in W_{new} but not in W_{old} . A new write was introduced.

The same four-way classification applies to external call records, keyed on the target address expression.

Canonicalization Structural equivalence of symbolic expressions requires canonicalization, because independently executed symbolic runs produce syntactically different terms for the same logical operation. Before any comparison (of keys, values, or path constraints), we apply a shared set of tree-shape normalizations that target known sources of syntactic divergence:

- *Solver artifacts.* Collapse internal storage-materialization chains to their base array; normalize boolean encoding ($\text{NotEquals}(a, b) \rightarrow \text{Negate}(\text{Equals}(a, b))$); eliminate double negation; strip execution-specific identifiers (solver-assigned variable names, execution counters).
- *Comparison direction.* Rewrite asymmetric comparisons to a canonical form ($\text{BVUGT}(a, b) \rightarrow \text{BVULT}(b, a)$, and similarly for signed variants), so that semantically identical comparisons have a unique representation.
- *Compiler redundancies.* Strip identity operations ($\text{bvmul}(x, 1) \rightarrow x$; $\text{bvdiv}(x, 1) \rightarrow x$); remove redundant address masks ($\text{BVAnd}(0\text{xFF} \dots \text{FF}_{160}, x) \rightarrow x$); strip zero-extension from 160 to 256 bits; fold dead conditionals ($\text{if}(\text{equal}(c, c), \text{dead}, \text{live}) \rightarrow \text{live}$); fold constant sub-expressions including KECCAK256 with concrete arguments.

- *Operand ordering.* Sort children of commutative operations (`BVAdd`, `BVMul`, `Equals`, `BVAnd`, `BVOr`) by canonical tree shape to eliminate operand-order artifacts from different compiler versions.

Key Matching Storage keys in the EVM are 256-bit words, often derived from nested `KECCAK256` hash over a base slot number and one or more mapping keys [157]. For example, the expression `balances[msg.sender]` compiles to `SHA3(CALLER||0x00)`, where `0x00` is the base slot of the `balances` mapping. We resolve symbolic key expressions by tracing their SHA3 derivation structure: for a key of the form `SHA3( $k_1 || k_2$ )`, we recursively extract the base slot and key arguments. Two keys are structurally equivalent if, after canonicalization, their SHA3 derivation trees have the same shape and their base slot numbers match.

Multi-Path Handling A function may traverse multiple execution paths, each producing different sets of storage-write records. We handle this in two steps. First, *constraint intersection*: for each key, we collect the path constraints from all paths that write to that key and retain only constraints that hold on *every* writing path. A constraint present on all paths represents a true behavioral invariant; constraints on a subset of paths reflect branch-specific conditions and are excluded to avoid false positives from branch-order noise. Second, *value cross-matching*: within a key-matched group, we compare individual (k, v) pairs across versions rather than selecting a single representative. If the old version writes

values $\{a, b\}$ at a given key and the new writes $\{a\}$, the comparison correctly classifies a as matched and b as only-old.

4.3.4 Path Constraint Comparison

Beyond comparing the *effects* of a function (its storage writes and external calls), we also analyze whether the *conditions* under which those effects occur differ between versions. A function that writes to the same storage slots with the same values, but under weaker path constraints, represents a security regression: operations previously restricted to the contract owner, for instance, may become executable by arbitrary callers.

Three-Pass Matching After applying the canonicalization described in §4.3.3, we compare old and new constraint sets in three successive passes, ordered from cheapest to most expensive. Canonicalization alone resolves the majority of cases; the SMT-backed passes handle residual semantic equivalences that syntactic normalization cannot capture.

Pass 1: Exact shape match. Pair constraints whose canonical tree shapes are identical. This requires only structural comparison and handles the majority of cases.

Pass 2: SMT equivalence. For constraints that remain unpaired after Pass 1, query the solver for logical equivalence ($o_i \Leftrightarrow n_j$). This catches semantic equivalences that syntactic normalization cannot resolve, such as algebraically equivalent expressions involving different intermediate variables.

Pass 3: SMT implication. For old constraints still unmatched, check whether any new constraint logically implies the old one, i.e., whether $n_j \wedge \neg o_i$ is unsatisfiable. If so, the new constraint is *strictly stronger* than the old: the operation is guarded by a tighter condition in the new version. Such entries are annotated as *strengthened* rather than flagged as regressions.

Old constraints that remain unmatched after all three passes represent genuine behavioral differences: conditions enforced in the old version that are absent or weakened in the new.

Semantic Classification Unmatched constraints are classified by the semantic role of their constituent variables. Constraints involving `CALLER` or `ORIGIN` indicate changes to access-control logic. Constraints derived from `SLOAD` expressions indicate changes to storage-based guards, such as checks on mapping values, reentrancy locks, or pause flags. This classification determines which detector processes the finding (§4.3.5).

4.3.5 Detection

The comparison results from the preceding stages are consumed by a suite of detectors organized around three classes of behavioral divergence. Each class targets a distinct type of security-relevant change.

Sensitive Slot. Not all storage slots are equally security-relevant: a slot that stores an administrator address or an initialization flag has far greater impact when corrupted than

one that caches a derived value or tracks a nonce. Inspired by the impact analysis of Ruaro et al. [141], we classify a slot as sensitive if it satisfies any of three conditions: (1) it participates in access-control decisions, as identified by the decompiler’s guard analysis (e.g., the slot holding the contract owner’s address); (2) every write to the slot occurs inside an access-controlled block, indicating that the value is protected by privileged logic (e.g., fee parameters, oracle addresses); or (3) it is read but never written by any function in the current implementation, representing deployment-time configuration set by the constructor or a prior version’s initializer (e.g., a pool address, a token-decimals parameter, or an initial owner set during proxy deployment). Detectors use this classification to focus on changes that affect security-critical state rather than incidental bookkeeping.

Class 1: New Reachability The most basic signal: an operation (storage write, external call) is reachable in the new version but has no counterpart in the old. This captures the majority of real-world upgrade vulnerabilities, including both accidental authorization omissions and deliberate backdoors.

(C1a) *Only-new call.* An external call record (CALL, DELEGATECALL) appears in the new version’s behavioral profile but is classified as *only-new* by the comparison of §4.3.3: no call record in the old version’s profile has a structurally matching target expression. This occurs when the old version either reverted before reaching the call, executed a different set of calls, or contained no call instructions in the function’s reachable code. Ex-

ternal calls can transfer value or delegate control to arbitrary code, making new, unvetted calls a primary risk. Findings are annotated with the call target’s concreteness: symbolic or zero-address targets indicate higher risk than concrete non-zero addresses, which more commonly arise from library or facade refactoring.

(C1b) *State revival*. The old implementation reverts on every feasible path (zero non-reverting states, zero storage writes), while the new implementation executes successfully and writes to at least one sensitive storage slot. This indicates that a previously unreachable function has become callable after the upgrade, typically because a guard flag no longer evaluates as expected. The transition from “always reverts” to “writes sensitive state” is a strong indicator of re-initialization or administrative-state-reset vulnerabilities.

(C1c) *Critical slot change*. A static pre-analysis identifies *critical* storage slots: those whose values influence whether an ERC-20 token operation executes (e.g., a balance or allowance mapping that feeds into a branch condition guarding `transfer`, `transferFrom`, or `approve`). When a storage write to a critical slot appears in the new version but not in the old, the upgrade introduced a new state mutation on a slot that directly governs token movement.

Two additional Class 1 detectors (C1d and C1e) target newly added functions through the static analysis track described in §4.3.6. These detectors address the same conceptual class (new reachability) but do not need to operate on symbolic execution results.

Class 2: Weakened Conditions When both versions produce records at the same storage key or call target, we compare the path constraints under which those operations execute. The key insight is that a function may write to the same slot with the same value, yet be more permissive in the new version because a guard condition was dropped or relaxed. Using the three-pass constraint comparison of §4.3.4 (structural matching, SMT equivalence, and SMT implication), we identify old-version constraints that have no equivalent or stronger counterpart in the new version. Such unmatched constraints represent conditions that the old version enforced but the new version does not, widening the set of states under which the operation can execute. We classify these by the semantic role of their constituent variables:

(C2a) Weakened auth constraint. All access-control constraints (involving `CALLER` or `ORIGIN`) present in the old version’s constraint set are absent from the new. The function’s authorization requirement was fully removed.

(C2b) Weakened storage constraint. One or more constraints derived from storage reads (`SLOAD`) present in the old version’s path constraints have no match in the new. This captures both the complete removal and the partial weakening of storage-based guards, such as message-validation checks, replay-prevention flags, or pause mechanisms.

Class 3: Value Divergence When both versions write to the same storage key under comparable constraints, but the value expressions differ, the function computes a different

result. We classify the structural relationship between the old and new value expressions and flag three high-signal categories: *same-leaves-different-operators* (the same symbolic inputs are combined through a different formula), *constant-to-symbolic* (a previously hard-coded value became dependent on runtime inputs), and *fully-different* (the old and new expressions share no symbolic leaves at all). The remaining categories (same structure, constant-to-constant, symbolic-to-constant) represent benign drift from different on-chain state at the simulation blocks and are informational.

(C3a) Value divergence. A storage write classified as *changed* (same key, different value) targets a sensitive slot, and the value-expression structural class falls into one of the three high-signal categories above. This captures formula rewrites (e.g., a fee calculation that changes from a hardcoded percentage to a storage-derived parameter), input substitutions (e.g., reading from a different mapping index), and complete computation replacements.

Pair-Level Filters After per-function detection, pair-level filters suppress false positives arising from systematic, non-security-relevant upgrade patterns.

Call-target rotation. When a contract upgrade redirects internal module calls to new addresses (e.g., pointing a lending pool at a new interest-rate model), every function that references the old module produces both an only-new and an only-old call finding. If three or more selectors within the same upgrade pair exhibit this symmetric pattern, the changes

are consistent with a library or facade migration rather than a targeted modification; call-related findings on such pairs are suppressed.

Timestamp drift. The old and new implementations are simulated at adjacent blocks (block $b-1$ and $b+1$), so expressions involving `block.timestamp` concretize to slightly different values. When both sides of a changed storage write are concrete constants within the plausible Unix timestamp range and differ by at most 600 seconds, the finding is classified as an execution-environment artifact and suppressed.

4.3.6 Static Analysis for Newly Added Functions

Newly added functions have no direct baseline (no function with the same selector existed in the old version), so the differential symbolic execution track we described above cannot be applied. However, for access-control-related properties, a useful compromise is possible: rather than comparing the function against a prior version of itself, we compare its storage footprint and guard structure against *other functions that operate on the same state*. If every existing function that writes to a given storage slot enforces an access control check, a newly added function that writes to the same slot without such a check is suspicious.

This cross-function comparison operates on the TAC callgraph and the decompiler’s mapping from variables to their known constant values, requiring no symbolic execution.

Storage-Construct Matching Each function's `SSTORE` and `SLOAD` key expressions are resolved to their base mapping slots by tracing through `KECCAK256` derivation chains using the decompiler's constant-value mappings. A base slot is a small constant (< 256) that anchors a mapping or direct-access storage variable. Two functions with overlapping base slots access the same logical storage constructs and are natural comparison targets.

Guard Signal Extraction For each function, the presence of a caller-based access control check is determined by identifying a `CALLER` opcode followed by an equality comparison and a conditional jump within the same basic block. This heuristic distinguishes functions that verify the caller's identity (e.g., owner-only modifiers) from functions that merely read the caller address for other purposes.

Detection Checks Two Class 1 detectors are applied to each newly added function that writes to at least one storage slot.

(C1d) Broken guard lifecycle. Many contract operations are meant to execute only once: a reward claim should mark the reward as claimed, an initialization function should set an initialized flag, a vote should record that the voter has voted. The common pattern is to read a guard slot, check its value, perform the operation, then write the guard slot to prevent re-execution. If the newly added function reads the guard but never writes it back, the guard is checked but never consumed, and the operation can be repeated indefinitely.

We detect this by identifying storage slots whose values feed into branch conditions (SLOAD into JUMPI within the same basic block) in the newly added function but are never written back by any of its reachable code paths. To confirm that the slot is state that was meant to be mutated rather than read-only configuration, we verify that at least one function in the old implementation wrote to the same slot. Finally, we require that the function writes to at least one sensitive slot, filtering out guards on non-security-relevant paths.

(C1e) Unguarded admin write. The newly added function writes to a storage slot that, in the old implementation, was written *exclusively* by access-controlled functions, yet the new function itself has no access control. A new unguarded write path to a previously protected slot represents a direct privilege escalation. We consider a function access-controlled if it contains either a caller-identity check (CALLER compared in a branch condition) or a storage-based guard (reading a sensitive slot in a branch condition, such as an initialization flag). This definition naturally accommodates proxy-style initializers that use a one-time storage guard rather than a caller check, without requiring pattern-matching heuristics on function signatures.

4.4 Evaluation

We evaluate DISSONANCE along three dimensions: (1) whether it detects known upgrade vulnerabilities (ground-truth validation), (2) what it finds on a corpus of real-world upgrades (discovery run), and (3) how precise its findings are (triage and false-positive analysis).

4.4.1 Datasets

Incident Dataset We curate a ground-truth dataset of 12 publicly documented smart contract upgrade incidents, drawn from online post-mortems. Table 4.1 lists the incidents, which span 2021–2024 across Ethereum mainnet and Binance Smart Chain. Each entry records the proxy address, old and new implementation addresses, the block number of the upgrade transaction, and the vulnerable function selector.

Upgrade-Pair Dataset We derive our upgrade pairs following the upgradable proxy identification method of ProxyEx [204] via Datalog-based bytecode analysis. More specifically, for ease triage, we extract consecutive implementation pairs per proxy, retain only those where both implementations have Etherscan-verified source, sample one transition per proxy chain, and remove mass-migration duplicates, producing 2,733 upgrade pairs (2018–2024). Deduplicating by (old_impl, new_impl) yields 1,713 unique pairs.

4.4.2 Ground-Truth Validation

Of the 12 incidents, two are not detected. Ankr’s vulnerability resides inside an ECDSA signature verification routine whose recursive byte-processing causes solver state explosion before the symbolic executor reaches the vulnerable mint operation. Euler Finance’s `donateToReserves` is a newly added function that modifies reserve balances without calling the `checkLiquidity` post-condition that every other reserve-modifying function enforces. As a newly added function, it is routed to the static analysis track. The function does contain guards (it reads a storage slot in a branch condition) and does write to storage, but neither guard lifecycle nor access control is broken: the vulnerability is a missing post-operation invariant check, which is beyond the checks we can model in the static track.

DISSONANCE detects the remaining 10 incidents. Table 4.2 summarizes the results. Each incident is detected by at least one detector, and several are detected by multiple detectors independently.

The two analysis tracks are complementary. The symbolic execution track detects six incidents through behavioral differencing of modified functions: Compound (reward-index value divergence from a comparison-operator bug), DeusDAO (allowance parameter swap), Shata Capital (storage layout collision), Pike (re-initialization via storage byte-offset shift), Meerkat (backdoor function with new external call), and Nomad (message-

validation guard removal). The static track detects five incidents through structural analysis of newly added functions: SafeMoon and GymNet (unguarded writes to admin-protected state), Meerkat (backdoor initialization, lacks access control), and Level (claim function that reads but never invalidates an epoch guard). One incident (Meerkat) is detected by both tracks: its backdoor `out(address)` function triggers `C1a` (new external call) on the symbolic track and `C1e` (unguarded admin write) on the static track.

Case Study: Compound The Compound incident illustrates value divergence on a critical code path. Proposal 62 split the `compSpeeds` mapping into separate borrow and supply rates, initializing the supply index for each market to a base value `compInitialIndex = 1036`. When a user claims rewards, `distributeSupplierComp` is supposed to initialize the user's stored `supplierIndex` to match the current `supplyIndex` if it has not been set. The initialization guard checks:

```
1 if (supplierIndex == 0 && supplyIndex > compInitialIndex)
2     supplierIndex = compInitialIndex;
```

Immediately after migration, `supplyIndex` equals `compInitialIndex` (not greater), so the strict `>` comparison fails and `supplierIndex` remains zero. The subsequent reward computation then executes:

$$\Delta = \text{supplyIndex} - \text{supplierIndex} = 10^{36} - 0$$

inflating the accrued reward by orders of magnitude. Had the guard used `>=`, the initialization would succeed, `supplierIndex` would be set to `1036`, and the delta would correctly

be zero. Both old and new versions write to the same key (`compAccrued[user]`), but the value expressions diverge: the old version computes a normal reward delta, while the new version computes against the uninitialized zero index. This is a Class 3 finding: the same storage write produces a different value due to a single-character comparison-operator error.

4.4.3 Discovery Run

We run DISSONANCE on all 1,713 upgrade pairs using 20 parallel workers, each with a 300-second per-function time cap and a 20 GB memory limit. The per-pair time cap is 3,600 seconds. A memory watchdog monitors each worker’s proportional set size (PSS) and terminates workers that exceed the limit, writing an error result so the row is skipped on resume.

Of 1,713 pairs, 1,356 complete successfully (79.2%), and 259 terminate with errors (memory exhaustion, decompiler failures, or solver assertion violations).

DISSONANCE flags 164 pairs (12.1% of completed pairs) with at least one finding. Table 4.3 shows the per-detector breakdown.

DISSONANCE discovers 2 previously unreported vulnerabilities in the discovery dataset, both detected by the static analysis track’s C1e (unguarded admin write) detector: a confirmed on-chain rug pull and a broken access-control regression.

FeesManager: Confirmed On-Chain Rug Pull The FeesManager proxy manages fee computation for the FBM token, a small-cap ERC-20 on Ethereum. The upgrade introduced two coordinated malicious changes. First, `estimateFees()` gained a `require(tx.origin == owner())` gate. Because the FBM token calls `estimateFees()` internally during every swap, this change froze all trading for non-owners. Second, a new function `mintTokens()` was added with no access control:

```
function mintTokens() public {  
    fbmContract.feesManagerCancelBurn(  
        0x49fe..., 1000000000000000000000000);  
}
```

Despite its name, `feesManagerCancelBurn` invokes `mint` internally, creating tokens from nothing. With the FBM token using 6 decimals, the raw amount corresponds to approximately 10^{16} tokens, inflating the supply by a factor of 10^{10} over the original $\sim 954,000$.

On-chain records confirm the attack was executed: `mintTokens()` was called 17 hours after the upgrade, minting the inflated supply to the owner's address. The deployer wallet subsequently routed funds through Tornado Cash. Cle (unguarded admin write) detects this because `mintTokens()` is a newly added function that writes to storage slots previously accessible only through the `onlyOwner`-gated `updateFbmAddress`, yet contains no caller check.

Instadapp: Broken Access Control The Instadapp Aave V2 automation contract allows users to register their DeFi Smart Accounts (DSAs) for automated health-factor rebal-

ancing on Aave. The legitimate registration function, `submitAutomationRequest`, binds to `msg.sender`:

```
function submitAutomationRequest(  
    uint256 safeHF, uint256 thresholdHF  
) external onlyDSA(msg.sender) { ... }
```

The upgrade added `submitAutomationRequestMock`, which replaces `msg.sender` with a caller-supplied `user_` parameter. The `onlyDSA(user_)` modifier verifies that `user_` is a valid DSA but does not verify that the caller owns or is authorized to act on behalf of that DSA. Any account can therefore register any DSA for automation with attacker-chosen health-factor thresholds, enabling forced position rebalancing (with associated fees and swap slippage borne by the victim) without the DSA owner’s consent.

Three pieces of evidence confirm this was unintentional: the function is named “Mock” (a testing artifact), it was removed in a subsequent upgrade, and it was never called on-chain during its deployment lifetime. Cle (unguarded admin write) detects this because the function writes to the `_userAutomationConfigs` and `_userLatestId` mappings, which the old implementation wrote only through the `msg.sender`-gated `submitAutomationRequest`.

4.4.4 Triage and Precision

To evaluate the precision of DISSONANCE’s findings, we perform a structured triage of all flagged pairs across both analysis tracks: 117 symbolic execution findings and 70 static

findings (after sensitive-slot gating). For each finding, we read the verified Solidity source code of both implementations, identify the flagged function, compare the old and new versions, and assign a verdict according to the following taxonomy:

- **TP** (true positive): the finding corresponds to a confirmed vulnerability, through independent analysis that verifies the bug and its security impact.
- **Lim-FP** (limitation false positive): the finding identifies a real behavioral difference, but the change is benign (refactoring, feature addition with intact guards). This class is a fundamental limitation of the analysis, which cannot distinguish intentional changes from regressions.
- **Spur-FP** (spurious false positive): a detector misfire caused by a simulation artifact, comparator imprecision, or compilation difference. These are engineering imprecisions that can be addressed with better normalization, not fundamental analysis limitations.

Triage Results Table 4.4 summarizes the triage results across both analysis tracks on the discovery set. Across both tracks, we identify 2 discovery TPs, 100 lim-FPs, and 87 spur-FPs.

Both discovery TPs are found by C1e (unguarded admin write) on the static track: the FeesManager rug pull and the Instadapp broken access control. No discovery TPs

arise from the symbolic execution track, where all findings are either lim-FPs (intentional feature changes, refactors) or spur-FPs (compiler drift, non-auth caller usage).

The symbolic execution track’s 37 spur-FPs are dominated by compiler and optimizer drift and `msg.sender` used as a data parameter rather than an access-control gate. The static track’s 50 spur-FPs arise primarily from two patterns: functions that read an admin or configuration slot for access control but correctly never write it (the *auth-guard read* pattern), and functions that access dynamic arrays, triggering a bounds check on the array-length slot that the detector misclassifies as a consumable guard. The unguarded-admin-write detector’s spur-FPs are dominated by non-CALLER authorization mechanisms (ECDSA signatures, cross-chain message authentication) that the detector’s CALLER-based heuristic does not recognize.

4.5 Discussion

4.5.1 Limitations

Path Explosion and Memory Contracts with deeply nested data structures, reflection-token fee arithmetic, or recursive byte-processing routines (e.g., ECDSA verification, TypedMemView parsing) cause solver state explosion that exhausts the per-worker memory limit. Approximately 15% of pairs in our dataset trigger this condition. The Ankr

incident exemplifies the structural limit: recursive signature verification produces thousands of solver states before reaching the vulnerable mint operation, making the function unanalyzable within practical resource bounds.

Loop Bounding Symbolic execution bounds loop iterations at five by default. This is sufficient for most contract patterns (mapping lookups, array iteration over small collections) but misses behaviors that manifest only after many iterations, such as cumulative rounding errors in reward distribution loops. Raising the bound increases coverage at the cost of exponential state growth.

4.5.2 Specific False-Positive Patterns

Several recurring patterns deserve specific mention, as they inform both the limitations of the approach and potential directions for improvement.

Gas Optimization Drift A common Solidity optimization shifts storage guards from `!= 0` to `> 1`, leaving one wei in the slot to avoid the 15,000-gas cost of a zero-to-nonzero `SSTORE`. The constraint-comparison pass sees an operator change (`NotEquals` to `BVUGT`) and a constant change (0 to 1), producing a structural mismatch. While the guard is technically weaker (`> 1` admits fewer values than `!= 0`), the difference is economically meaningless. Detecting this pattern would require value-aware constraint comparison, which the current tree-shape matcher does not perform.

Optimizer-Toggle Drift When two implementations compile from identical Solidity source but with different optimizer settings (e.g., optimizer enabled versus disabled, or different `runs` values), the resulting bytecode can differ substantially. The opcode-bag filter (§4.3.1) catches cases where the instruction set is identical but block layout differs. However, optimizer-on versus optimizer-off can introduce or remove identity operations (`bvmul(x, 1)`, dead conditionals), producing genuinely different opcode bags. The source-identity filter addresses the subset of these cases where verified source is available.

4.6 Related Work

Smart Contract Vulnerability Detection A large body of work has focused on identifying vulnerabilities in smart contracts through static analysis, symbolic execution, and fuzzing. Oyente [100] was among the first tools to apply symbolic execution to EVM bytecode, targeting vulnerabilities such as integer overflows and transaction-ordering dependence. Mythril [34] extended this approach with broader vulnerability coverage, while ETHBMC [53] employed bounded model checking for more precise reasoning. On the static analysis side, Securify [173] encodes security properties as Datalog patterns over contract dependency graphs, and Slither [49] provides a fast, pattern-based static analysis framework widely adopted in practice. Ethainter [16] performs composite vulnerability detection via information-flow analysis, and Sailfish [15] targets state-inconsistency bugs

by analyzing storage-access patterns. Fuzzing-based approaches, including Echidna [64], ConFuzzius [170], and EFCF [140], generate transaction sequences to trigger violations of user-specified or automatically inferred properties. More recent work has combined techniques: SmarTest [155] uses language models to guide symbolic execution, and ILF [71] learns fuzzing strategies from symbolic execution traces. These tools primarily target individual contract deployments and focus on well-known vulnerability classes (e.g., reentrancy, integer overflow). In contrast, DISSONANCE targets a fundamentally different problem: detecting behavioral regressions introduced by contract *upgrades*, which requires comparing two versions of an implementation rather than analyzing a single one.

Upgradeable Smart Contract Security The proxy-based upgrade pattern, standardized through EIP-1967, allows developers to replace the logic of a deployed contract while preserving its state and address. Several recent works have studied the prevalence and security implications of this pattern. Bodell et al. [14] propose a taxonomy of eleven upgrade patterns and develop USCHunt, an automated framework that identified over 2,500 upgrade-related security issues across 861K contracts on eight blockchains. Li et al. [92] develop USCDetector to identify upgradeable contracts without source code and conduct a large-scale empirical study of over 60 million Ethereum contracts. Salehi et al. [147] evaluate six upgradeability patterns and characterize the upgrade behavior of 1.4 million proxy contracts. CRUSH [141] specifically targets storage collision vulnerabilities by

grouping all contracts behind a proxy (including successive implementations) and checking for slot-level type mismatches across them (e.g., one implementation treating a slot as an `address` while another treats it as a mapping base). These works operate at the *structural* level (pattern classification, storage layout type compatibility). DISSONANCE is complementary: it operates at the *behavioral* level, symbolically executing both versions and comparing their state-modifying operations. This enables detection of semantic regressions that structural analysis cannot capture, such as weakened access controls, unauthorized fund transfers, missing replay guards, and sub-slot storage shifts that change runtime values without altering slot-level types. Conversely, CRUSH can flag type collisions on code paths that symbolic execution may not reach due to solver timeouts or path explosion, making the two approaches complementary layers in an upgrade security pipeline.

Differential and Regression Program Analysis Comparing program versions to identify behavioral changes has a long history in software engineering. Semantic Diff [78] was an early system for summarizing the observable input-output differences between two procedure versions. Person et al. [128] introduced *differential symbolic execution*, which characterizes behavioral differences by partitioning the symbolic state space of two program versions and analyzing divergent paths. DiSE [129] extends this by combining static change-impact analysis with symbolic execution to focus only on behaviors affected by

code changes. SymDiff [87] provides language-agnostic semantic differencing for imperative programs via the Boogie intermediate verification language, while Lahiri et al. [88] further developed *differential assertion checking* to verify that assertions preserved in old versions are not violated by new ones. In the formal methods community, Godlin and Strichman [59] established *regression verification*, which proves equivalence of closely related program versions rather than verifying each against a full specification. DISSONANCE adapts the core insight of differential analysis to the EVM domain: rather than comparing source-level semantics, it operates on bytecode through symbolic execution of the TAC intermediate representation, comparing storage writes and external calls across versions. The two-track strategy (symbolic execution for modified functions, static cross-function analysis for newly added functions) is motivated by the practical constraints of analyzing thousands of real-world upgrades at scale.

Invariant Detection for Smart Contracts Dynamic invariant detection, pioneered by Daikon [43, 45], discovers likely program invariants from execution traces. This idea has been adapted to the smart contract domain by several recent works. InvCon [96] dynamically infers contract invariants from transaction histories and was later extended with static verification to produce formally verified invariants [97]. Trace2Inv [23] demystifies the effectiveness of invariants for securing smart contracts and proposes methods to improve their precision. SmartInv [180] leverages multimodal learning and foundation models to

infer invariants, uncovering 119 zero-day vulnerabilities. While these systems derive invariants from a *single* contract version’s execution history, DISSONANCE performs cross-version comparison: it uses symbolic execution to extract the behavioral profile of each version and then detects regressions by structurally differencing these profiles.

4.7 Conclusion

We presented DISSONANCE, a bytecode-level differential analysis approach that detects security-relevant behavioral changes in smart contract upgrades by combining symbolic execution of modified functions with static analysis of newly added functions. On 12 known incidents, DISSONANCE detects 10; on 1,713 unknown upgrade pairs, it discovers 2 previously unreported vulnerabilities: a confirmed on-chain rug pull and a broken access-control regression.

Table 4.1: Ground-truth incident dataset.

#	Incident	Chain	Date	Loss	Type
1	Meerkat Finance	BSC	2021-03	\$31M	new
2	Compound	Mainnet	2021-09	\$150M	mod
3	GymNetwork	BSC	2022-06	\$2.1M	new
4	Nomad	Mainnet	2022-08	\$190M	mod
5	Ankr [†]	BSC	2022-12	\$5M	mod
6	Thoreum	BSC	2023-01	\$680K	mod
7	Shata Capital	Mainnet	2023-02	\$5.1M	mod
8	Euler [†]	Mainnet	2023-03	\$197M	new
9	SafeMoon	BSC	2023-03	\$9M	new
10	DeusDAO	Mainnet	2023-05	\$6.5M	mod
11	Level Finance	BSC	2023-05	\$1M	new
12	Pike Finance	Mainnet	2024-04	\$1.6M	mod

[†] Not detected. Type: new = newly added function,

mod = modified function

Table 4.2: Ground-truth detection results. C1: new reachability. C2: weakened conditions. C3: value divergence.

Incident	C1 symbolic			C1 static		C2		C3
	C1 _a	C1 _b	C1 _c	C1 _d	C1 _e	C2 _a	C2 _b	C3 _a
Compound								✓
DeusDAO			✓				✓	
Shata	✓						✓	
Pike		✓						
Nomad							✓	
SafeMoon				✓	✓			
Meerkat	✓				✓			
Level				✓				
GymNet				✓	✓			
Thoreum				✓				

Table 4.3: Discovery-run findings by detector.

Class	Track	Detector	Findings	Pairs
C1	Symbolic	Only-new call	40	31
	Symbolic	State revival	16	7
	Symbolic	Critical slot change	33	22
	Static	Broken guard lifecycle	60	42
	Static	Unguarded admin write	33	28
C2	Symbolic	Weakened auth	9	9
	Symbolic	Weakened storage	39	31
C3	Symbolic	Value divergence	30	17
Total			260	164*

* Unique pairs; individual rows overlap.

Table 4.4: Triage results by detector and track, excluding ground-truth re-detections. Counted as pairs.

Level	Track	Detector	TP	Lim	Spur
C1	Symbolic	Only-new call	0	22	9
	Symbolic	State revival	0	5	2
	Symbolic	Critical slot	0	17	5
	Static	Broken guard	0	11	31
	Static	Unguarded admin	2	9	19
C2	Symbolic	Weakened auth	0	5	4
	Symbolic	Weakened storage	0	17	14
C3	Symbolic	Value divergence	0	14	3

Chapter 5

Conclusions and Future Work

This dissertation introduced the notion of *retrospective artifacts* and demonstrated that the traces naturally left behind by software development, deployment, and evolution are a practical and effective source of behavioral reference for security analysis.

BRAN mines development process metadata to predict likely vulnerable areas in large codebases, discovering real bugs in the Linux kernel when used to guide fuzzing. HOUSTON builds behavioral models from DeFi protocol transaction histories, demonstrating superior detecting capabilities while functioning in real time. DISSONANCE compares successive smart contract versions across upgrades, catching suspicious behavioral changes in the unguarded gaps.

Across two domains and three artifact types, the core methodology persists. The results show that retrospective analysis is a coherent security methodology that complements conventional code-based approaches.

5.1 Future Directions

Combining artifact classes. Each system in this dissertation exploits a single class of retrospective artifact. A natural extension is to combine multiple classes in a single analysis, for instance jointly modeling execution history and version lineage to better detect anomalous transactions when DeFi protocols grow.

LLM-assisted retrospective analysis. Large language models present a significant opportunity for retrospective security analysis. LLMs can reason over commit messages, code diffs, and developer discussions to extract richer semantic signals from development metadata than hand-crafted features alone. They can summarize and interpret anomalous execution traces in natural language, improving the explainability of alerts. And they can assist in differential analysis by understanding the intent behind code changes, distinguishing deliberate refactors from accidental security regressions. As LLMs continue to improve at code understanding, integrating them into retrospective analysis pipelines is a promising direction that could substantially improve both the coverage and usability of the systems presented in this dissertation.

Bibliography

- [1] Apache flink project. <https://flink.apache.org/>. (Used version: 1.6.0).
- [2] Cppcheck - a tool for static c/c++ code analysis.
- [3] Flashbots. <https://www.flashbots.net/>.
- [4] flawfinder - lexically find potential security flaws ("hits") in source code.
- [5] Metamask: The ultimate crypto wallet for defi, web3 apps, and nfts. <https://metamask.io/>.
- [6] Pscan, <http://deployingradius.com/pscan/>, 2017.
- [7] Rats, <https://code.google.com/archive/p/rough-auditing-tool-for-security/>, 2017.
- [8] and and. Improving vulnerability prediction accuracy with secure coding standard violation measures. In *2016 International Conference on Big Data and Smart Computing (BigComp)*, pages 115–122, Jan 2016.
- [9] M. J. Bach et al. *The design of the UNIX operating system*, volume 5. Prentice-Hall Englewood Cliffs, NJ, 1986.
- [10] R. Behnke. Explained: The Pike Finance hack (April 2024). <https://www.halborn.com/blog/post/explained-the-pike-finance-hack-april-2024>, 2024.
- [11] P. E. Black. SARD. <https://samate.nist.gov/SARD/>.
- [12] blocksecteam. Revest finance vulnerability. <https://blocksecteam.medium.com/revest-finance-vulnerabilities-more-than-re-entrancy-1609957b742f>, 2022.
- [13] BNB Chain. BNB chain. <https://www.bnbchain.org>, 2024.

- [14] W. E. Bodell III, S. Meisami, and Y. Duan. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
- [15] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 161–178. IEEE, 2022.
- [16] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 454–469. ACM, 2020.
- [17] BSC. Bnb chain. <https://www.bnbchain.org/en/bnb-smart-chain>, 2025.
- [18] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36:7346–7354, 05 2009.
- [19] CertiK. Bacon protocol incident analysis. <https://www.certik.com/resources/blog/bacon-protocol-incident-analysis>, 2022.
- [20] chainsec. Timeline of defi exploits. <https://chainsec.io/defi-hacks/>.
- [21] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), 2009.
- [22] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang. SODA: A generic online detection framework for smart contracts. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.
- [23] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long. Demystifying invariant effectiveness for securing smart contracts. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering*, 2024.
- [24] B. Chernis and R. Verma. Machine learning methods for software vulnerability detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, IWSPA '18*, pages 31–39, New York, NY, USA, 2018. ACM.
- [25] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *J. Syst. Archit.*, 57(3):294–313, Mar. 2011.

Bibliography

- [26] cisa.gov. Wannacry ransomware attack. https://www.cisa.gov/sites/default/files/FactSheets/NCCIC%20ICS_FactSheet_WannaCry_Ransomware_S508C.pdf, May 2017.
- [27] CNBC. Defi is the wild-west of crypto, 2021.
- [28] Coinbase. Euler compromise investigation. <https://www.coinbase.com/blog/euler-compromise-investigation-part-1-the-exploit>, 2023.
- [29] Coindesk. Axie infinity ronin network suffers 625m exploit, 2022.
- [30] CoinDesk. Nomad bridge drained of nearly \$200 million in exploit. <https://www.coindesk.com/tech/2022/08/02/nomad-bridge-drained-of-nearly-200-million-in-exploit/>, 2022.
- [31] Coindesk. Nomad bridge drained of nearly 200-million in explploit, 2022.
- [32] Cointelegraph. Euler finance attack how it happened and what can be learned, 2023.
- [33] Cointelegraph. Euler finance hacked despite 10 audits in 2 years. <https://cointelegraph.com/news/euler-finance-hacked-despite-10-audits-in-2-years-says-ceo>, 2023.
- [34] ConsenSys. Mythril. <https://github.com/ConsenSys/mythril>, 2022.
- [35] T. Cook, A. Latham, and J. H. Lee. Dappguard: Active monitoring and defense for solidity smart contracts. 2017.
- [36] Debian. Debian - the universal operating system. <https://www.debian.org/>.
- [37] DeFiLlama. Dashboard. <https://defillama.com/chain/Ethereum>, 2024.
- [38] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, (2):222–232, 1987.
- [39] D. Drysdale. Coverage-guided kernel fuzzing with syzkaller. *Linux Weekly News*, 2:33, 2016.
- [40] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. 01 2019.

- [41] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu. Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In *IJCAI*, 2019.
- [42] dune.com. Dune. <https://dune.com/>, 2025.
- [43] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, 1999.
- [44] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, 2000.
- [45] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.
- [46] Ethereum. Ethereum. <https://ethereum.org/en/>, 2022.
- [47] Ethereum. Erc-20 token standard. <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/>, 2023.
- [48] evmdecompiler. evmdecompiler. <https://evmdecompiler.com/>, 2025.
- [49] J. Feist, G. Grieco, and A. Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019.
- [50] Forbes. Terrorists, north korea and other illicit actors move beyond bitcoin, 2023.
- [51] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE, 1996.
- [52] Forta. What is forta firewall: Redefining onchain security, 2025.
- [53] J. Frank, C. Aschermann, and T. Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774, 2020.
- [54] W. Fu and T. Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 72–83, New York, NY, USA, 2017. ACM.

- [55] Y. Gai, L. Zhou, K. Qin, D. Song, and A. Gervais. Blockchain large language models. *arXiv preprint arXiv:2304.12749*, 2023.
- [56] A. Ghaleb, J. Rubin, and K. Pattabiraman. Etainter: Detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 728–739. ACM, 2022.
- [57] GitHub. GitHub. <https://github.com>.
- [58] GitHub. Github’s public apis. <https://developer.github.com/v3/>. (Accessed: 2019-01-15).
- [59] B. Godlin and O. Strichman. Regression verification: Proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.
- [60] Google. syzbot. <https://syzkaller.appspot.com/>.
- [61] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019.
- [62] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Analyzing the out-of-gas world of smart contracts. *Commun. ACM*, 63(10):87–95, 2020.
- [63] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis. Elipmoc: advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 6:1–27, 2022.
- [64] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 557–560. ACM, 2020.
- [65] A. Griffith. *GCC: the complete reference*. McGraw-Hill, Inc., 2002.
- [66] 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 *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1793–1810, 2023.

- [67] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct 2005.
- [68] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 232–241, New York, NY, USA, 2006. ACM.
- [69] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, Nov 2012.
- [70] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, M. W. McConley, J. M. Oppen, S. P. Chin, and T. Lazovich. Automated software vulnerability detection with machine learning. *CoRR*, abs/1803.04497, 2018.
- [71] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 531–548. ACM, 2019.
- [72] hexagate. Linea x hexagate: Building the world's most secure 12. <https://www.hexagate.com/blog/linea-hexagate-building-the-worlds-most-secure-12>, 2025.
- [73] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.
- [74] T. Hu, X. Liu, T. Chen, X. Zhang, X. Huang, W. Niu, J. Lu, K. Zhou, and Y. Liu. Transaction-based classification and detection approach for ethereum smart contract. *Information Processing & Management*, 58(2):102462, 2021.
- [75] hypernative. Hypernative to bolster on-chain security across the linea network. <https://www.hypernative.io/blog/hypernative-to-bolster-on-chain-security-across-the-linea-network>, 2025.
- [76] Immunefi. Hack analysis: Beanstalk governance attack. <https://medium.com/immunefi/hack-analysis-beanstalk-governance-attack-april-2022-f42788fc821e>, 2022.

- [77] ironblocks. ironblocks. <https://ironblocks.com/>, 2025.
- [78] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance (ICSM 1994)*, pages 243–252, 1994.
- [79] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE’07)*, pages 96–105, May 2007.
- [80] W. S. Journal. How north korea’s hacker army stole \$3 billion in crypto, funding nuclear program, 2023.
- [81] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.
- [82] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.
- [83] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473, 2008.
- [84] B. Krebs. Experts Urge Rapid Patching of Struts Bug, 2018. <https://krebsonsecurity.com/2018/08/experts-urge-rapid-patching-of-struts-bug/>.
- [85] C. Krügel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 201–208, 2002.
- [86] J. Krupp and C. Rossow. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.
- [87] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification, 24th International Conference (CAV 2012)*, pages 712–717. Springer, 2012.

- [88] S. K. Lahiri, K. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*, pages 345–355, 2013.
- [89] J. R. Larus, T. Ball, , R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May 2004.
- [90] ledgerwatch. Erigon. <https://github.com/ledgerwatch/erigon>, 2023.
- [91] J. Li and M. D. Ernst. Cbcd: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 310–320, June 2012.
- [92] X. Li, J. Yang, J. Chen, Y. Tang, and X. Gao. Characterizing ethereum upgradable smart contracts and their security implications. In *Proceedings of the ACM Web Conference 2024*, 2024.
- [93] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang. Sysevr: A framework for using deep learning to detect software vulnerabilities. *CoRR*, abs/1807.06756, 2018.
- [94] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *CoRR*, abs/1801.01681, 2018.
- [95] J. Liu, Y. Chen, B. Tan, I. Dillig, and Y. Feng. Learning contract invariants using reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–11, 2022.
- [96] Y. Liu and Y. Li. Invcon: A dynamic invariant detector for ethereum smart contracts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.
- [97] Y. Liu, C. Zhang, and Y. Li. Automated invariant generation for solidity smart contracts. *IEEE Transactions on Software Engineering*, 2025.
- [98] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundness: A manifesto. *Commun. ACM*, 58(2):44–46, Jan. 2015.

- [99] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.
- [100] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 254–269. ACM, 2016.
- [101] X. Lyu, M. Zhang, X. Zhang, J. Niu, Y. Zhang, and Z. Lin. An empirical study on ethereum private transactions and the security implications. *arXiv preprint arXiv:2208.02858*, 2022.
- [102] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. {DR}. {CHECKER}: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1007–1024, 2017.
- [103] MakerDAO. The maker protocol: Makerdao's multi-collateral dai (mcd) system. <https://makerdao.com/en/whitepaper/>, 2023.
- [104] R. Malhotra. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.*, 27(C):504–518, Feb. 2015.
- [105] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
- [106] I. Medeiros, N. Neves, and M. Correia. Dekant: A static analysis tool that learns to detect web application vulnerabilities. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 1–11. ACM, 2016.
- [107] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.
- [108] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44, 2014.

- [109] B. Meyer. Soundness and Completeness: With Precision, 2019. <https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext>.
- [110] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [111] S. Moshtari, A. Sami, and M. Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013:8–17, 05 2013.
- [112] S. Moshtari, A. Sami, and M. Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013(5):8–17, 2013.
- [113] L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, and L. Zhang. Building program vector representations for deep learning. *CoRR*, abs/1409.3358, 2014.
- [114] J. C. Munson and S. G. Elbaum. Code churn: a measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31, Nov 1998.
- [115] N. Mutual. Analysis of the barley finance exploit. <https://medium.com/neptune-mutual/analysis-of-the-barley-finance-exploit-d2df61b98c80>, 2024.
- [116] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07*, pages 529–540, New York, NY, USA, 2007. ACM.
- [117] V. H. Nguyen and L. M. S. Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics, MetriSec '10*, pages 3:1–3:8, New York, NY, USA, 2010. ACM.
- [118] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 18*, page 653–663. ACM, 2018.
- [119] nluedtke. Linux kernel cves. <https://github.com/nluedtke/linux.kernel.cves>. (Accessed: 2019-01-15).

- [120] node real. bsc-erigon. <https://github.com/node-real/bsc-erigon>, 2025.
- [121] OpenAI. Gpt-4.1. OpenAI API, Apr. 2025. Large language model; context window 1 million tokens; API access.
- [122] openrefactory. The battle of the bugs. https://www.openrefactory.com/intelligent-code-repair-icr/#_ftnref2, 2020.
- [123] S. Ozkan. Cve details: The ultimate security vulnerability datasource. <https://www.cvedetails.com>. (Accessed: 2019-01-15).
- [124] B. Pan, N. Stakhanova, and Z. Zhu. Ethershield: Time interval analysis for detection of malicious behavior on ethereum. *ACM Trans. Internet Technol.*, 2023.
- [125] paradigmxyz. Blazing-fast implementation of the ethereum protocol. <https://github.com/paradigmxyz/reth>, 2025.
- [126] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.
- [127] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.
- [128] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008)*, pages 226–237, 2008.
- [129] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, 2011.
- [130] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10*, pages 447–456, New York, NY, USA, 2010. ACM.
- [131] phylax. Verifiable, embedded hack prevention. <https://phylax.systems/>, 2025.

- [132] pmccabe. McCabe-style function complexity and line counting. <https://people.debian.org/~bame/pmccabe/pmccabe.1>. (Used version: 2.6).
- [133] K. Qin, S. Chaliasos, L. Zhou, B. Livshits, D. Song, and A. Gervais. The blockchain imitation game. *arXiv preprint arXiv:2303.17877*, 2023.
- [134] K. Qin, Z. Ye, Z. Wang, W. Li, L. Zhou, C. Zhang, D. Song, and A. Gervais. Towards automated security analysis of smart contracts based on execution property graph. *arXiv preprint arXiv:2305.14046*, 2023.
- [135] K. Qin, L. Zhou, Y. Afonin, L. Lazzaretti, and A. Gervais. Cefi vs. defi—comparing centralized to decentralized finance. *arXiv preprint arXiv:2106.08157*, 2021.
- [136] QuillAudits. Decoding euler finance’s \$197 million exploit — quillaudits, 2023.
- [137] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics. *Inf. Softw. Technol.*, 55(8):1397–1418, Aug. 2013.
- [138] Rekt. Rekt, leaderboard. <https://rekt.news/leaderboard/>, 2022.
- [139] rekt.news. Overcompensated. <https://rekt.news/overcompensated/>, 2021.
- [140] M. Rodler, D. Paaßen, W. Li, L. Bernhard, T. Holz, G. Karame, and L. Davi. Efcf: High performance smart contract fuzzing for exploit generation. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 449–471, 2023.
- [141] N. Ruaro, F. Gritti, R. McLaughlin, I. Grishchenko, C. Kruegel, and G. Vigna. Not your type! detecting storage collision vulnerabilities in ethereum smart contracts. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium 2024*, 2024.
- [142] N. Ruaro, F. Gritti, R. McLaughlin, D. Meng, I. Grishchenko, C. Kruegel, and G. Vigna. A history of greed: Practical symbolic execution for Ethereum smart contracts. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2025.
- [143] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley. Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320, 2018.
- [144] Rust-lang. Rust language. <https://www.rust-lang.org/>, 2025.

- [145] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.
- [146] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 1157–1168, New York, NY, USA, 2016. ACM.
- [147] M. Salehi, J. Clark, and M. Mannan. Not so immutable: Upgradeability of smart contracts on ethereum. In *Financial Cryptography and Data Security Workshops (WTSC 2022)*, 2022.
- [148] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, Oct 2014.
- [149] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei. EThor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 621–640, 2020.
- [150] K. Security. The poly network attack, 2021.
- [151] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, Nov. 2011.
- [152] Y. Shin and L. Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '08*, pages 315–317, New York, NY, USA, 2008. ACM.
- [153] R. K. Shyamasundar. A framework of runtime monitoring for correct execution of smart contracts. In *Blockchain – ICBC 2022*, pages 92–116. Springer Nature Switzerland, 2022.
- [154] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.
- [155] S. So, S. Hong, and H. Oh. SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution.

- In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1361–1378. USENIX Association, 2021.
- [156] soliditylang. Contract abi specification. <https://docs.soliditylang.org/en/develop/abi-spec.html>, 2023.
- [157] soliditylang. Storage layout. https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html, 2023.
- [158] Soliduslabs. Rug pull crypto scams. <https://www.soliduslabs.com/post/rug-pull-crypto-scams>, 2022.
- [159] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: Finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11*, pages 1069–1084, New York, NY, USA, 2011. ACM.
- [160] J. Stephens, K. Ferles, B. Mariano, S. Lahiri, and I. Dillig. Smartpulse: Automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571, 2021.
- [161] L. Su, X. Shen, X. Du, X. Liao, X. Wang, L. Xing, and B. Liu. Evil under the sun: understanding and discovering attacks on ethereum decentralized applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1307–1324, 2021.
- [162] T. Sun, N. He, J. Xiao, Y. Yue, X. Luo, and H. Wang. All your tokens are belong to us: Demystifying address verification vulnerabilities in solidity smart contracts. *arXiv preprint arXiv:2405.20561*, 2024.
- [163] SunWeb3Sec. Defihacklabs. <https://github.com/SunWeb3Sec/DeFiHackLabs>, 2025.
- [164] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th Conference on Security Symposium, SS'08*, pages 379–394, Berkeley, CA, USA, 2008. USENIX Association.
- [165] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014, Proceedings*, pages 109–129. Springer, 2014.

- [166] S. Team. Solidity. <https://soliditylang.org>, 2022.
- [167] V. Team. Vyper. <https://vyperlang.org>, 2022.
- [168] thestreet.com. Crypto stolen in 2024 tops 1.2 billion. <https://www.thestreet.com/crypto/markets/crypto-stolen-in-august-declines-but-2024-losses-already-top-1-2-billion>, 2024.
- [169] TornadoCash. Tornadocash. <https://tornado.cash/>, 2025.
- [170] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119, 2021.
- [171] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, page 664–676. ACM, 2018.
- [172] C. F. Torres, M. Steichen, and R. State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1591–1607, 2019.
- [173] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 67–82, 2018.
- [174] Uniswap. Uniswap info. <https://v2.info.uniswap.org/home>, 2023.
- [175] J. Vanegue and S. K. Lahiri. Towards practical reactive security audit using extended static checkers. In *2013 IEEE Symposium on Security and Privacy*, pages 33–47, May 2013.
- [176] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267, Dec 2000.
- [177] D. Vyukov. gcc: add fuzzing coverage, 2016. <https://codereview.appspot.com/280140043>.

- [178] D. Vyukov. Recommended kernel configs, 2019. https://github.com/google/syzkaller/blob/master/docs/linux/kernel_configs.md.
- [179] S. Wang, C. Zhang, and Z. Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.
- [180] S. J. Wang, K. Pei, and J. Yang. SmartInv: Multimodal learning for smart contract invariant inference. In *45th IEEE Symposium on Security and Privacy (S&P 2024)*, 2024.
- [181] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt. SoK: Decentralized finance (DeFi). In *IEEE Symposium on Security and Privacy (S&P)*, 2023.
- [182] Wikipedia. Rice theorem. https://en.wikipedia.org/wiki/Rice%27s_theorem, 2024.
- [183] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.
- [184] S. Wu, D. Wang, J. He, Y. Zhou, L. Wu, X. Yuan, Q. He, and K. Ren. Defi-ranger: Detecting price manipulation attacks on defi applications. *arXiv preprint arXiv:2104.15068*, 2021.
- [185] Y. Xue, J. Fu, S. Su, Z. A. Bhuiyan, J. Qiu, H. Lu, N. Hu, and Z. Tian. Preventing price manipulation attack by front-running. In *International Conference on Artificial Intelligence and Security*, pages 309–322. Springer, 2022.
- [186] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.
- [187] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.
- [188] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

- [189] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.
- [190] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM.
- [191] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 797–812, Washington, DC, USA, 2015. IEEE Computer Society.
- [192] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. ACM.
- [193] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 344–353, Nov 2017.
- [194] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 157–168, New York, NY, USA, 2016. ACM.
- [195] ycharts. Ethereum market cap. https://ycharts.com/indicators/ethereum_market_cap, 2024.
- [196] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.
- [197] C. Zengler and W. Kuchlin. Encoding the linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, volume 2010, pages 51–56, 2010.

- [198] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 309–320, New York, NY, USA, 2016. ACM.
- [199] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2775–2792, 2020.
- [200] Y. Zhang, W. Yu, Z. Li, S. Raza, and H. Cao. Detecting Ethereum Ponzi Schemes Based on Improved LightGBM Algorithm. *IEEE Transactions on Computational Social Systems*, 9(2):624–637, 2022.
- [201] Z. Zhang, Z. Lin, M. Morales, X. Zhang, and K. Zhang. Your exploit is mine: Instantly synthesizing counterattack smart contract. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1757–1774, 2023.
- [202] Z. Zhang, Z. Lin, M. Morales, X. Zhang, and K. Zhang. Your exploit is mine: Instantly synthesizing counterattack smart contract. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1757–1774. USENIX Association, 2023.
- [203] Z. Zhang, B. Zhang, W. Xu, and Z. Lin. Demystifying exploitable bugs in smart contracts. In *Proceedings of the 45th International Conference on Software Engineering*, 2023.
- [204] C. Zheng et al. ProxyEx: Proxy contract detection and upgradeable analysis at ecosystem scale. In *International Conference on Software Engineering (ICSE)*, 2025.
- [205] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2444–2461. IEEE, 2023.
- [206] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu. How far we have progressed in the journey? an examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(1):1, 2018.