Universidad Nacional
de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Fuzzing Ethereum Smart Contracts

Luis Alberto López Alvar

Director: Luis de la Torre Cubillo

Trabajo de Fin de Máster

Máster Universitario
en Ingeniería de Sistemas y Control

Septiembre de 2025

Agradecimientos

Quisiera dedicar este trabajo, en primer lugar, a la UNED, por ofrecer la oportunidad de continuar la formación académica a quienes, por circunstancias personales, no lo tendríamos fácil en otros contextos. A mi director, Luis de la Torre Cubillo, por su apoyo constante, su guía y el interés mostrado durante todo este proceso.A mi familia, por su paciencia y comprensión, especialmente por el tiempo que no he podido compartir con ellos. Y a Andreea, por su apoyo, comprensión y compañía en este recorrido.

Resumen

Este trabajo investiga el estado del arte del fuzzing y su aplicación a los contratos inteligentes en Ethereum, un ámbito en el que las vulnerabilidades pueden ocasionar pérdidas económicas significativas. Aunque el fuzzing ha demostrado ser eficaz en la detección de ciertos errores en software tradicional, su aplicación a contratos inteligentes presenta desafíos particulares. A diferencia de los programas convencionales, los contratos son altamente dependientes de su estado y suelen exponer vulnerabilidades que no se manifiestan como fallos de ejecución que terminan con la finalización anómala del software, sino como fallos lógicos. Este trabajo aborda estas limitaciones mediante una revisión de la literatura sobre fuzzing general y fuzzing específico de Ethereum, identificando retos persistentes así como potenciales espacios de investigación. Para complementar este análisis, se llevó a cabo un experimento práctico extendiendo una herramienta existente con dos estrategias diseñadas para mitigar la explosión de estados. Los resultados preliminares sugieren que estas técnicas permiten equilibrar la eficiencia con la exploración del espacio de estados. Pese a su alcance limitado, la tesis aporta tanto una síntesis de la literatura como una contribución práctica, señalando posibles líneas de investigación futura para mejorar el fuzzing en Ethereum en particular y el fuzzing en general.

Palabras clave: Fuzzing, Ethereum, Contratos Inteligentes, Explosión de Estados, Pruebas de Seguridad, Detección de Vulnerabilidades

Abstract

This work investigates the state of the art of fuzzing and its application to Ethereum smart contracts, a domain where vulnerabilities can lead to significant financial losses. While fuzzing has proven effective at uncovering memory errors and crashes in traditional software, its application to smart contracts presents unique challenges. Unlike conventional programs, smart contracts are highly stateful and often expose vulnerabilities that do not manifest as crashes but rather as logical flaws. This work addresses these limitations by reviewing the literature on general fuzzing and Ethereum-specific fuzzing, identifying persistent challenges and research gaps. To complement this review, a practical experiment was conducted by extending an existing tool with two strategies designed to mitigate state explosion. The preliminary results suggest that these techniques can balance efficiency with the exploration of states space. Despite its limited scope, the thesis contributes both a synthesis of the literature and a modest practical addition, highlighting future research directions to improve fuzzing in Ethereum in particular and fuzzing in general.

Keywords: Fuzzing, Ethereum, Smart Contracts, State Explosion, Security Testing, Vulnerability Detection

Glossary

Black-Box Fuzzing Fuzzing with no knowledge of program internals, generating random inputs blindly.

Code Coverage A measure of how much of a program's code has been executed during testing.

Ethereum A decentralized blockchain platform that enables the deployment of smart contracts and decentralized applications.

Ethereum Virtual Machine (EVM) The runtime environment that executes Ethereum smart contracts.

Formal Verification The mathematical proof that software satisfies its specification in all possible cases.

Fuzzing An automated testing method that feeds programs many unexpected inputs to uncover flaws or vulnerabilities.

Grey-Box Fuzzing Fuzzing that uses feedback like code coverage to guide input generation more effectively.

Smart Contract A self-executing program on a blockchain that enforces an agreement when conditions are met.

State Explosion The rapid growth of possible program states, making exhaustive exploration infeasible.

Static Analysis Examining program code without execution to detect potential flaws or vulnerabilities.

Symbolic Execution Running programs with symbolic inputs to explore multiple paths using logical constraints.

Vulnerability A weakness in software that can be exploited to cause unintended behavior or security issues.

White-Box Fuzzing Fuzzing that applies program analysis and symbolic execution to systematically explore execution paths.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smart contracts, first conceptualized by Szabo (1996), gained practical realization with Ethereum's launch in 2015 Wood (2014), which built on the decentralized ledger concepts initially introduced by Bitcoin. Ethereum introduced a programmable and decentralized environment, enabling diverse applications beyond cryptocurrency transactions. Since its emergence, Ethereum-based smart contracts have rapidly expanded into various sectors, notably decentralized finance (DeFi). This widespread adoption has resulted in contracts frequently managing high-value digital assets, making cybersecurity integral to the blockchain ecosystem. Notable incidents, such as the DAO attack in 2016 Mehar et al. (2017) (which caused a hard-fork) and more recent incidents like the exploits observed in early 2025 Rekt-News (2025), highlight ongoing vulnerabilities and the high stakes involved.

Security practices targeting smart contracts have matured significantly, incorporating advanced methodologies such as formal verification and comprehensive bug bounty programs. Despite these rigorous measures, vulnerabilities persist, continuously pushing the boundaries of cybersecurity research in blockchain technologies. Emerging innovations like post-quantum cryptography Buterin (2024) and privacy-preserving protocols (e.g., Zero-Knowledge proofs) also introduce new complexities and potential vulnerabilities, further emphasizing the need for advanced security testing methodologies.

Fuzz testing, commonly known as fuzzing, is one of the most effective automated techniques for vulnerability discovery. Originally introduced as a technique involving random input generation aimed at triggering anomalous program behaviors or crashes, fuzzing has evolved dramatically, now integrating sophisticated techniques such as machine learning (ML) and Large Language Models (LLMs) to improve effectiveness and coverage.

Ethereum has become a primary focus for fuzz testing research due to its extensive use, its Turing-complete Ethereum Virtual Machine (EVM), and the significant economic value managed by its smart contracts. Ethereum's specific properties, such as immutability, public accessibility, permissionless interactions, and high-value asset management, amplify the importance and complexity of robust security testing methodologies.

Nevertheless, existing fuzzing techniques applied to Ethereum face notable challenges,

including limited ability to thoroughly explore stateful contract behaviors, the complexity of interactions between multiple contracts, and scalability issues arising from blockchain-specific data structures and storage mechanisms. Addressing these limitations is crucial for enhancing smart contract security.

1.1 Motivation

Smart contracts exhibit inherent security risks due to unique blockchain properties, such as immutability, openness, asset value, and computational completeness (Wei et al., 2025, p. 2). Ethereum, the dominant platform for smart contract deployment, inherently amplifies these security concerns, making robust testing methodologies, particularly fuzzing, critically important.

Moreover, Ethereum's widespread adoption has created a standardization effect through the EVM, in the blockchain ecosystem, enabling fuzzing methodologies developed for Ethereum to be relatively easily adapted to other EVM-compatible blockchains. Conversely, non-EVM-compatible blockchains, such as Solana, or those employing unique architectures like Oasis's Trusted Execution Environment (TEE), introduce substantially different design complexities, thereby limiting the direct applicability of Ethereum-focused fuzzing techniques.

Thus, this research explicitly focuses on fuzzing within Ethereum and general EVM-compatible contexts, excluding consideration of distinctly different architectures such as Solana and Oasis, while acknowledging their importance for future studies.

1.2 Research Questions and Objectives

Understanding the current state of the art of fuzzing and Ethereum fuzzing is key to then be able to introduce novel tecniques, methods or improve the current existing results. For this research focus into answer the following Main Research Questions:

- **RQ1:** What are the current general limitations and challenges in fuzz testing?
- **RQ2:** Specifically, what limitations currently exist when fuzz testing smart contracts?
- **RQ3:** Within Ethereum and EVM-compatible smart contracts, what specific limitations do existing fuzzing techniques face?
- **RQ4:** What effective fuzzing techniques commonly employed in broader software testing are still underexplored or unutilized in the context of smart contracts, and vice versa?

Additionally the following Secondary Research Questions are also interesting for potential future directions as well as understanding the in between state of the art of fuzzing in smart contracts:

- **SRQ1:** Do current fuzzing methodologies adequately cover the specific features of EVM-compatible blockchains?
- **SRQ2:** What is the status and maturity of fuzz testing methodologies within prominent non-EVM blockchains?

Finally as an output of this answers some initial practical proposals on how to improve some of the current results are tried in order to gain also practical understanding of the current state of the art for future directions and implementations of improvements.

1.3 Organization of the thesis

This thesis is structured as follows. Chapter 2 introduces the research methodology adopted to address the research questions and outlined in Chapter 1. Chapter 3 starts with a general context of Ethreum smart contract security introducing the vulnerabilities and the security assessment methods, among which fuzzing is one of the most relevant. Then provides a review of the state of the art in fuzzing and Ethereum-specific fuzzing, starting from general fuzzing to then focus in Ethereum fuzzing, establishing the bases for the subsequent experiment. The design of the experiment is presented in Chapter 4, while Chapter 5 reports and discusses the results obtained from the experiments. Finally, Chapter 6 synthesizes the findings, answers the research questions, and outlines conclusions together with potential directions for future work.

Chapter 2

Methodology

The methodology adopted in this work consisted of two main stages: (i) the collection of existing literature on fuzzing in general and smart contract fuzzing in particular, and (ii) a structured review and analysis of these works in order to understand the state of the art (SOTA) and identify open challenges and research gaps.

First we systematically collected prior work from both academic and non-academic sources. On the academic side, we use primarly Google Scholar and Scopus. A set of queries have been designed for this. On the non-academic side, we also incorporated grey literature due to its significant influence in the fuzzing community. For this purpose, we searched web engines and platforms such as GitHub (for tools and implementations) and X/Twitter (for community-driven discussions and announcements). Including these sources is particularly relevant because many impactful fuzzing frameworks and experimental prototypes are released directly as open-source projects without being accompanied by peer-reviewed publications.

Search Query
("security" OR "fuzzing" OR "static analysis" OR "formal verification" OR "symbolic execution" OR "vulnerability") AND ("web3" OR "smart contract" OR "blockchain")

Table 2.1: Search Queries used in Scopus for smart contract fuzzing

Second, when analyzing the collected works, we applied a differentiated strategy depending on the domain. For general fuzzing, we conducted a *non exhaustive review*, prioritizing existing surveys and critical contributions that illustrate the evolution of the field from its origins to the current SOTA. This approach allowed us to focus on identifying historical trends, methodological shifts, and emerging research directions rather than reviewing every individual contribution in detail. In contrast, for smart contract fuzzing we performed a more *exhaustive review*, systematically analyzing each available work. The rationale is based on the different maturity of the two areas: while the general fuzzing literature is vast and

established, the research on smart contract fuzzing is still comparatively small, making a comprehensive review feasible and necessary.

This methodological choice already revealed one important insight: the overwhelming majority of both academic and non-academic works on smart contract fuzzing target Ethereum, with other blockchain platforms receiving only anecdotal coverage. In practice, this means that "smart contract fuzzing" is often synonymous with "Ethereum fuzzing" leaving significant gaps in other ecosystems.

The outcome of this methodology has two primarly benefits. On one hand, it provides a structured understanding of the trends, challenges, and gaps in both fuzzing and smart contract fuzzing. On the other hand, it enables us to select one research problem that sits at the intersection of the two domains: a problem that represents both a current trend and an area strongly affected by existing limitations. This problem then serves as the focal point for our experimental work, where we attempt to evaluate the implications of tackling it in practice.

Chapter 3

State of the Art

3.1 Ethereum Smart Contract Security

Ethereum smart contract vulnerabilities can be viewed across three fundamental layers: the Solidity language, the Ethereum Virtual Machine (EVM), and the underlying blockchain architecture Atzei et al.. This layered perspective, originally proposed by Atzei et al. together with the recent initiative "Trillion Dollar Security" from The Ethereum Foundation Ethereum Foundation (2025), highlights that bugs may originate from high-level code (Solidity), low-level execution (EVM), or the blockchain's design and protocol rules. In addition to these technical layers, it is useful to consider the smart contract lifecycle. An Ethereum smart contract passes through stages such as development (writing and testing the code), pre-deployment (finalizing code before launch), and post-deployment (the contract running on-chain). Each phase introduces distinct security concerns for example, coding mistakes during development or new threat vectors emerging after deployment as the contract interacts with other on-chain components Ethereum Foundation (2025); Areta Research (2025). It is also worth noting that the Ethereum ecosystem's security is not confined to smart contracts alone. Vulnerabilities can arise from off-chain or Web2 components (like wallets, front-end dApps, or APIs) that interface with Ethereum Blockchain Security Standards Council, Inc. (2025); Ethereum Foundation (2025); Wang and Team (2024); Raza (2025). Industry standards are emerging to address these broader attack surfaces, for instance, the Blockchain Security Specification Stack proposed by the BSSC covers node operations, key management, token security, and general security principles across blockchain systems Blockchain Security Standards Council (2025), but in this work we focus strictly on vulnerabilities within smart contracts and their lifecycle.

3.1.1 Common Vulnerabilities

Smart contract vulnerabilities have been extensively studied to understand their root causes and to develop taxonomies for classification. Two early community-driven resources were the

Decentralized Application Security Project (DASP) Top 10 Wong and Hemmel (2018) and the Smart Contract Weakness Classification (SWC) registry Diligence (2019). The DASP 2018 list enumerated the top 10 highest-risk smart contract vulnerability types, while the SWC registry catalogued 36-37 specific smart contract weaknesses. These provided a starting point for vulnerability classification, but they are now somewhat outdated. Recent research has updated these categorizations and introduced new taxonomies. For example Wei et al. (2025) present an updated list of top 4 smart contract vulnerability classes (superseding the old DASP list), and industry practitioners have developed standards like the Smart Contract Security Verification Standard (SCSVS) OWASP Foundation (2025) or frameworks such as the Enterprise Ethereum Alliance's EthTrust Security Levels Enterprise Ethereum Alliance (2023)

Common vulnerabilities in Ethereum contracts include well-known patterns such as Reentrancy (unexpected recursive calls enabling theft of funds), Broken Access Control (lack of proper permission checks), Arithmetic Bugs (e.g., integer overflow/underflow errors), Gas-Related Vulnerabilities (issues with gas costs or limit that can break contract logic), and various Economic or Logic Flaws that attackers exploit for, among others, financial gain. These categories appear across many taxonomies and are frequently cited in surveys. Researchers have observed that there is no single authoritative taxonomy – different studies often use different names or groupings for similar issues Zhou et al. (2022). For instance, the bug class commonly known as "integer overflow" is universally recognized, but more complex logical flaws might be categorized differently by different authors. This inconsistency in terminology and classification makes it challenging to unify vulnerability research Zhou et al. (2022).

Academic work continues to expand the understanding of vulnerability patterns. A recent comprehensive study by Zhang et al. (2023) introduced the notion of "machine-auditable bugs" (MUBs) – vulnerabilities that can be systematically detected by automated tools – and catalogued 306 such bugs. They grouped these into seven broad categories: price oracle manipulation, erroneous accounting, identifier uniqueness violations, inconsistent state updates, privilege escalation, atomicity violations, and implementation-specific bugs. This classification captures many nuanced DeFi vulnerabilities (for example, oracle manipulation and accounting errors are economic attack vectors common in financial contracts). Notably, Zhang et al. (2023) also found that the majority of real-world bugs are not easily detectable by machines – roughly 80% of exploitable smart contract bugs require human expertise to find, being too subtle or complex for current automated tools. This underscores that despite extensive catalogs of vulnerabilities, manual review and expert auditing remain critical in practice.

It is also observed that different domains of decentralized applications vary in their security maturity. For instance, Decentralized Finance (DeFi) contracts, which manage large amounts of value, have been subject to intensive scrutiny, multiple audits, and formal anal-

yses over the years. In contrast, newer application areas (for example, decentralized science) may not yet have the same level of community awareness or rigorous analysis. The vulnerability taxonomies and best practices continue to evolve as the Ethereum ecosystem changes. For instance, new Ethereum Improvement Proposals (EIPs) can introduce features with security implications. A recent example is EIP-7702, which proposes adding programmable logic to externally owned accounts (a step toward account abstraction); while it brings new functionality, it also opens up new attack surfaces and trust assumptions that developers must consider Nethermind Research (2025) Three Sigma (2025). Such protocol changes mean that the landscape of "common vulnerabilities" is not static – security researchers and auditors must continuously adapt to newly emerging patterns.

3.1.2 Security Assessment Methods

Over the years, a variety of security analysis methods have been developed to detect smart contract vulnerabilities. Different works sometimes use slightly different classifications for example,they separate fuzzing and symbolic execution as distinct categories, whereas other authors consider both under dynamic analysis Kezadri Hamiaz and Driss (2025); Zhang et al. (2023) , but broadly these methods include static analysis, dynamic analysis (including fuzz testing), formal verification, symbolic execution, and hybrid or AI-assisted approaches. Below we outline the main techniques and their characteristics:

Fuzzing Dynamic analysis methods involve executing the smart contract with various inputs to observe its behavior. Fuzzing is a prominent dynamic technique in which pseudo-random or systematically generated inputs are fed into the contract to see if any unexpected behaviors (such as reverts, integer overflow, or theft of funds) occur. Fuzzers like Contract-Fuzzer Jiang et al. (2018) generate many test transactions, sometimes guided by heuristics, to probe the contract's state space. The strength of dynamic analysis is that it observes actual runtime effects, making it good at catching vulnerabilities that manifest only when the contract is in a certain state or when multiple calls are made in sequence. The downside is that fuzzing can be time-consuming and computationally expensive It's impossible to exhaustively test all input combinations due to the enormous state space, so coverage is limited by how the inputs are chosen. Fuzzing may also overlook deep logical issues that require specific scenarios or attacker strategies not easily discovered by random testing. Despite these challenges, fuzzing has proven effective at finding bugs.

Static Analysis Static analysis involves examining the contract's code without executing it. The analyzer inspects the source code or compiled bytecode for patterns that indicate vulnerabilities. Techniques include control-flow analysis, data-flow analysis, and pattern matching. The advantage of static analysis is that it can catch issues early (at development or pre-deployment time). However, static analysis has significant limitations: because the

code isn't run, it cannot fully model runtime behaviors or interactions between contracts. This leads to many false positives (warnings about an issue that isn't truly exploitable) and false negatives (missing issues that occur only in specific runtime conditions). In other words, static analysis might over-approximate or under-approximate the program's actual execution. As a result, static tools often produce a large number of alerts, and developers must manually verify which ones are real concerns. Despite these drawbacks, static analysis is a staple in the early security review of contracts, as it's fast and can be automated as part of the development pipeline.

Formal Verification Formal verification provides mathematical assurance that a contract satisfies its specification in all possible executions. In the context of smart contracts, the taxonomy established by Tolmach et al. (2021) divides it, when applied to smart contracts between contract-level approaches abstract interactions between contracts, users, and the blockchain, often applying model checking to temporal logic properties such as safety or liveness while program-level approaches reason directly about code or bytecode, using theorem provers (e.g., Coq, Isabelle, KEVM) or applied tools like the Certora Prover Certora Ltd. (2025), which have recently demonstrated the feasibility of applying formal verification in a more native Ethereum setting, offering a balance between automation and expressiveness. Despite its strong security guarantees, formal verification remains resource-intensive, requiring significant expertise and often abstract models, which limits its broader adoption.

Symbolic Execution Symbolic execution is a hybrid approach (sometimes considered a form of dynamic analysis, other times its own category) where the contract is executed with symbolic inputs instead of concrete values. In a symbolic execution engine (such as hevm argotorg (2025) or Greed Gritti et al. (2023); Ruaro et al. (2024)), input variables are treated as symbols (unknowns) and the engine explores the program's execution paths by reasoning about logical conditions (path constraints) rather than specific numbers. Whenever a branch (e.g., an 'if' statement) depends on an input, the engine can conceptually fork execution, exploring both true/false branches with the corresponding conditions added as constraints. By doing this, a symbolic executor can explore many possible behaviors of the contract in a single run. If it finds a path that leads to an error state (such as an assertion failure, or an observable violation like stealing Ether), it can often generate a concrete input (test case) that demonstrates the bug. The limitation is that symbolic execution struggles with deep paths or complex contracts because the number of paths grows exponentially (path explosion), and solving the constraint conditions can become very slow.

Importantly, no single method catches all bugs. Each technique has blind spots: fuzzing might miss edge-case logic flaws, static analysis might misjudge runtime-dependant behavior, formal methods might not scale to the entire code, etc. This is why in practice a combination of tools and manual review is used. In fact, as noted earlier, a large portion of vulnerabilities

(on the order of 80%) are not fully detectable by automated means Zhang et al. (2023). Human auditors play a vital role in examining business logic, composability issues, and novel attack vectors that tools might not recognize. Figure 2 in Zhang et al. (2023) illustrates that the majority of bugs in their study fell into the category of non machine auditable, meaning they were found through manual inspection or were of types not readily caught by existing scanners.

Given these realities, the Ethereum community has developed a security culture that goes beyond just running analysis tools. Smart contracts (especially those managing significant value) typically undergo multiple independent audits by professional security firms. Auditors use the aforementioned methods (static/dynamic analysis, etc.) alongside manual code review to find vulnerabilities. Often, audit reports are made public for transparency. In addition, many projects deploy with an accompanying bug bounty program, offering monetary rewards to security researchers who later discover any vulnerabilities that slipped through. These proactive measures have undoubtedly improved security, but they are not foolproof. High-profile failures have shown that even exhaustive auditing can miss critical bugs. For example, Cork Protocol suffered a $12 million exploit in 2025 despite having passed four separate security audits and a bug bounty prior to launch Rekt.News (2025); Cork Protocol Docs (2025) .Similarly, a recent vulnerability in a Liquity-based DeFi project went undetected through multiple audits and was only found afterwards via targeted fuzz testing (“the bug that was missed”) nican0r (2025)

These cases highlight that security is a moving target and attackers often find novel ways to exploit subtle issues that auditors and tools did not anticipate.

Finally, beyond external audits and bounties, internal testing practices are an essential part of smart contract security. Responsible teams invest in thorough testing during development: unit tests for each function, integration tests for contract interactions, and even automated property-based tests and fuzzers integrated into their continuous integration (CI) pipelines . Mature projects treat their smart contracts with the same rigor (if not more) as traditional software, given the irreversibility of blockchain deployments. The Web3 development ecosystem has embraced practices like CI/CD with security checks, meaning that every code change is automatically tested and scanned. While the blockchain space has unique challenges, it is also at the forefront of secure software development life cycle practices, simply because the cost of failure is so high. In summary, ensuring Ethereum smart contract security requires a defense-in-depth approach: use automated tools (static analyzers, fuzzers, symbolic executors, etc.) to cover common issues, apply formal methods for critical components when feasible, have human experts audit the code, run bug bounties for continuous peer review, and maintain rigorous testing and monitoring throughout the contract’s life. This multi-faceted strategy is necessary to tackle the ever-evolving threat landscape of Ethereum smart contracts.

3.2 Fuzzing

Fuzzing testing or fuzz testing is an automated software testing technique that involves supplying a target program with large volumes of invalid, unexpected, or random inputs in order to trigger failures. The basic idea was introduced by Miller Miller et al. (1990) as a simple random testing of UNIX command-line programs. Early fuzzers operated in a black-box way, generating inputs blindly without any knowledge of program internals, proving to be effective at crashing programs and exposing bugs. Miller work and subsequent studies demonstrated that even simplistic random mutations could find numerous security vulnerabilities in software. This early success established fuzzing as a practical, fast method for bug hunting and the starting point for this technique.

3.2.1 SOTA of Fuzzing

After the idea introduced by Miller, fuzzing techniques evolved significantly. Researchers began to incorporate more intelligence and structure into input generation, giving rise to distinct categories of fuzzers. In general terms, fuzzers are now classified as **black-box**, **grey-box**, or **white-box** depending on how much program knowledge and analysis of the target software they employ. **Black-box** fuzzers remain close to the original model: they generate inputs without inspecting the program's code or state. Early black-box approaches include mutational fuzzers, which start from a few seed inputs and randomly mutate them (e.g., flipping bits or rearranging bytes), as well as generational fuzzers, which create inputs from scratch using a specification or grammar of the input format. Notable examples by the 2000s were generation-based fuzzers for network protocols and file formats (such as the word done by Oulu University Secure Programming Group (OUSPG) (1999–2001) from Oulu University) that used protocol definitions to produce well-formed test cases. While black-box fuzzers are simple and high-speed, their blind input generation often struggles with complex input validations, resulting in low code coverage. This limitation motivated the next stages in the evolution of fuzzing.

White-box fuzzing emerged in the mid-2000s as researchers applied program analysis to make fuzzing more effective incorporating more knowledge about the target program. White-box fuzzers leverage dynamic symbolic execution and constraint solving to explore program paths in a highly targeted way. Some of the first works that show this techniques, DART Godefroid et al. (2005) and SAGE Godefroid et al. (2012a,b), introduced the idea of analysing the program, gathering path conditions (branch predicates), and then solving those constraints to produce new inputs that drive execution down alternative paths. This means that instead of mutating inputs randomly, a white-box fuzzer treats input generation as a constraint satisfaction problem: for each observed program path, it computes the path's logical conditions and then modifies these conditions (e.g. negating a branch predicate) to solve for a new input that will follow a different path. This approach can systematically

force execution into hard-to-reach corners of the code, overcoming some of the "blindness" of black-box fuzzing. Notably, SAGE discovered numerous critical bugs in Windows and is credited with saving millions of dollars in software testing costs. White-box fuzzing, however, incurs heavy computational overhead due to constraint solving and may struggle with path explosion (the number of paths grows exponentially). By the 2010s, pure symbolic fuzzers like SAGE and KLEECadar et al. (2008) were mostly used on specific targets (e.g. libraries or drivers) rather than as general solutions. The quest to balance efficiency with deeper code coverage set the stage for the rise of grey-box fuzzing.

Grey-box fuzzing represents a tradeoff between whitebox and blackbox fuzzing that has become the dominant paradigm in recent years. Grey-box fuzzers typically execute the target program with lightweight instrumentation, obtaining feedback (such as code coverage or executed branches) to guide input mutations. This approach, exemplified by tools like AFL Zalewski (2013), libFuzzer Serebryany (2015), and Honggfuzz Świecki (2010), retains the speed and simplicity of random mutation-based fuzzing while using runtime feedback to evolve inputs that reach new program paths. A typical grey-box fuzzer will start with an initial corpus of seed inputs, mutate them in loops, and use instrumentation to detect if a mutation led to a new code path; if so, that input is saved for further fuzzing. This coverage-guided strategy, often implemented via edge coverage counters or similar feedback mechanisms, dramatically improves the efficiency of fuzzing by directing effort toward unexplored code. In this sense the introduction of AFLZalewski (2013) and its subsequent adoption in the field of fuzzing marked a turning point by showing that fuzzing can be more effective at finding deep bugs without the complexity of full symbolic analysis. Modern grey-box fuzzers incorporate a variety of heuristics (genetic algorithms for input selection, power schedules for allocation of fuzzing effort, etc.) and have been evolved with more sophisticated features like in-process fuzzing, persistent execution, and sanitizer integration to maximize the finding of bugs. This category of fuzzers has also been hybridized by integrating also constraint solving or taint analysis on specific cases showing the complexity of the area when it comes to establish taxonomies or categories for fuzzing.

The current SOTA of fuzzers is a product of this evolution and decades of improvements at many different dimensions. Currently fuzzing is not only applied to simple command-line utilities, but to many different applications, web browsers, network servers, file parsers, kernel drivers, embedded firmware, showing its versatility. In parallel another dimensions have appear and develop like infrastructures that improve the usage itself of the fuzzers. For example, Google OSS-Fuzz Google Open Source Security Team (2017) platform continuously runs grey-box fuzzers on hundreds of open-source projects, resulting in tens of thousands of bugs being discovered automatically.

Fuzzing has proved also its effectiveness and scalability have been demonstrated by empirical results. For instance, coverage-guided greybox fuzzing enabled Google to uncover

over 16.000 bugs in the Chrome browser Arya et al. (2019) and over 1.000 bugs across 47 projects just after five months Google Open Source Security Team (2017). These figures show how modern fuzzing tools, when deployed at scale, can continuously find bugs at an unprecedented rate. The 2016 DARPA Cyber Grand Challenge DARPA (2016a), a competition to build autonomous hacking systems, was won by Mayhem DARPA (2016b), a system heavily leveraging fuzzing (particularly symbolic fuzzing Brumley (2021)) to exploit and patch software flaws. In short, the current state of the art in fuzzing is one of broad deployment and significant impact.

Current SOTA is built on the foundational approaches (black-, grey-, white-box) but extend them with new features to tackle previous limitations. Fuzzers have incorporated the use of feedback-driven or coverage-guided fuzzing as the default strategy. For example, AFL and its successor AFL++ Fioraldi et al. (2020) use edge coverage instrumentation and genetic algorithms to optimize input mutations, continually expanding the set of program states reached. Variants like AFLFastBöhme et al. (2017a) and AFLGo Böhme et al. (2017b) introduced search strategies to prioritize rare paths or specific target locations. Other state-of-the-art grey-box fuzzers such as and Honggfuzz similarly use lightweight instrumentation and sophisticated mutation scheduling to maximize code coverage. Importantly, these fuzzers are not limited to trivial random mutations; they incorporate extensive heuristics (e.g., deterministic bit flips, arithmetic increments, dictionary replacement of tokens) and can detect various failure modes by integrating sanitizers for memory errors, assertion failures, etc.

Beyond general-purpose fuzzers, the state of the art also includes domain-specific and structure-aware fuzzing techniques designed for particular application domains or input formats. Researchers have recognized that incorporating knowledge of input structure can greatly enhance fuzzing of complex systems (e.g., compilers, network protocols, file parsers). Modern fuzzers often support grammar-based fuzzing, where inputs are generated according to a formal grammar or format specification instead of purely random bytes. For instance, tools like Restler Atlidakis et al. (2019) allowed testers to provide an input model. The advantage is higher throughput to deep program states since the fuzzer wastes less time on non well generated inputs. Another recent trend are frameworks for fuzzing: for example, FuzzFactory Padhye et al. (2019) generalizes coverage-guided fuzzing to support custom "feedback domains" allowing developers to plug in domain-specific feedback metrics and define waypoints – intermediate goals that guide the fuzzer toward complex conditions of interest. Using FuzzFactory, one can create specialized fuzzers for different goals without modifying the core fuzzing loop showing how fuzzing infrastructure has evolved to be highly customizable solutions.

Another prominent theme in state-of-the-art fuzzing is the integration of different hybrid fuzzing techniques like symbolic execution and. Pure black-box or grey-box fuzzing can get stuck when an input check (like a magic number or cryptographic checksum) blocks

further progress. Modern solutions often deploy a selective constraint solver to handle these problems. Tools like Angora Chen and Chen (2018) and QSYM dynamically switch to a lightweight symbolic execution mode when fuzzing encounters an input comparison that random mutation is unlikely to solve (e.g., a hard-to-guess 32-bit value). By solving for the exact bytes that satisfy the check, they generate new inputs that bypass this check and then resume regular fuzzing beyond that point. Similarly, taint analysis is used in some fuzzers (e.g., VUzzer Rawat et al. (2017), TaintScope to identify which parts of an input influence critical program states, so mutations can focus on those bytes.

In parallel, researchers are pushing fuzzing into new domains that were once considered out of scope. One example is kernel and OS fuzzing: fuzzers like Syzkaller Google and Trinity Jones generate sequences of system calls to crash operating system kernels. Syzkaller uses coverage guidance and a knowledge of system call APIs to systematically exercise deep kernel code, and it has found hundreds of Linux vulnerabilities. There are also efforts in fuzzing blockchain smart contracts as showed in the next section or to target AI systems. Additionally, state-of-the-art techniques include differential fuzzing (comparing outputs of different implementations for the same input to find inconsistencies, used for compilers or protocol implementations) and fuzzing in combination with model inference (learning a state machine of a protocol and then fuzzing transitions, etc.).

Finally, fuzzing research has begun to incorporate both machine learning (ML) and large language models (LLMs) to improve test generation. These approaches aim to automatically learn structural patterns from program executions or protocol specifications to produce more effective inputs. For instance, ChatAFL Meng et al. (2025) leverages LLMs to infer protocol grammars and guide stateful message generation, while NEUZZ She et al. (2019) employs an ML-based approach: it trains a neural network to approximate a program's branching behavior and then applies gradient descent on this learned model to craft inputs that flip specific branches, effectively solving complex conditions without requiring a formal solver.

3.2.2 Challenges and Research Gaps in Fuzzing

Despite its major successes, fuzzing is far from a solved problem. It faces persistent technical challenges and open research gaps. These can be grouped into several key areas:

- **Theoretical Foundation.** A persistent high-level research gap in fuzzing is the absence of a rigorous theoretical foundation. Unlike fields such as formal verification, fuzzing has primarily evolved through ad-hoc engineering practices and empirical trial-and-error. Frameworks like FuzzFactory Padhye et al. (2019), LibAFL Fioraldi et al. (2022), or Nyx Schumilo et al. (2021) demonstrate significant engineering progress enabling domain-specific fuzzing, modular and reusable infrastructures, and even scalable system-level fuzzing. Yet, these efforts remain tightly coupled to implementation concerns rather than based in a unifying theory. To date, there is no well-established model

that can, for example, estimate the fraction of bugs a given campaign will uncover, or determine how long fuzzing should run to yield statistically meaningful coverage. Without such foundations, research in the area remains fragmented: new frameworks and techniques regularly appear, but it is difficult to rigorously reason about why they succeed, or to compare them beyond empirical benchmarks.

- **Better Metrics and Benchmarks.** Another limitation is the lack of standardized metrics and benchmarks for evaluating fuzzers. Today, most studies report code coverage, raw crash counts, or executions per second. These metrics are useful but insufficient: coverage does not necessarily correlate with bug discovery, crash counts can influenced by duplicates or non-security bugs. The field lacks is a benchmark suite of diverse, real-world programs with known bugs of varying difficulty, enabling fair and reproducible comparisons. Furthermore, better metrics are needed to capture true effectiveness—such as vulnerability discovery rates or statistical measures of bug-finding significance. Even if projects like Google's FuzzBench Metzman et al. (2021) exist yet the gap remains open.

- **Scalable and Automated Fuzzing.** Although tools like AFL Zalewski (2013) and libFuzzer Serebryany (2015) lowered the entry barrier, fuzzing large or complex targets still requires significant expertise and manual setup. Fuzzing often demands reverse engineering, environment simulation, or custom instrumentation. This lack of automation is a clear research gap. Scalability is another challenge while companies some solutions like ClusterFuzz Arya et al. (2019) exist to run distributed fuzzing campaigns across thousands of cores, most research is needed in this direction to improve automatation, smarter orchestration of fuzzing resources, and modern frameworks to deal with fuzzers.

- **Enhanced Input Generation and Hybrid Techniques.** Handling highly structured inputs or stateful protocols remain an open limitation. Many programs reject malformed inputs, meaning blind mutations rarely exercise deep logic. Grammar-based and specification-aware fuzzing exist, but automatically inferring grammars or specifications is still immature. Similarly, hybrid fuzzing that combines fuzzing with symbolic or concolic execution has shown promise, but solvers often fail to scale on complex path conditions or cryptographic checks. Research is needed to make solver integration more lightweight and incremental. Another gap lies in stateful systems: fuzzing traditionally treats software as stateless, but real-world programs, protocols, GUIs, IoT devices maintain state across interactions. Developing methods to deal with states is not mature enough.

- **Advanced Bug Detection Oracles.** Fuzzers are still heavily biased toward detecting crashes and memory safety violations, leaving many other vulnerability classes under-

explored. Logical flaws, improper access controls, cryptographic misuse, errors often leave the program running smoothly, producing incorrect but non-crashing behavior that goes unnoticed by traditional fuzzing oracles. Developing more sophisticated oracles is therefore an active research need. Expanding the scope of bug detection is essential if fuzzing is to evolve from primarily finding memory corruption crashes to uncovering a bigger range of vulnerabilities.

3.3 Ethereum-Specific Fuzzing

Given Ethereum dominance in the smart contract ecosystem, fuzzing research has concentrated in the Ethereum Virtual Machine (EVM) and it's smart contracts ecosystem. This section outlines the main Ethereum fuzzing approaches, highlighting their contributions and the open challenges that still shape the field.

3.3.1 SOTA of Ethereum-Specific Fuzzing

The first Ethereum-focused fuzzer, ContractFuzzer Jiang et al. (2018), laid the groundwork by introducing fundamental mechanisms that later fuzzers would build upon. It performs offline EVM instrumentation, uses vulnerability oracles, and generates inputs based on function signatures. These core concepts—especially the use of oracles and EVM instrumentation—have remained integral to subsequent tools.

In the same year, ReGuard Liu et al. (2018) proposed a unique approach by translating smart contracts into C++ representations, enabling the use of traditional fuzzing frameworks such as AFL and libFuzzer. ReGuard specifically targeted reentrancy vulnerabilities, modeling them using a finite state machine and relying on runtime instrumentation for execution feedback.

In 2019 ILF He et al. (2019), proposes and hybrid approach that addresses core limitations of both traditional fuzzing and symbolic execution. In this sense they use symbolic execution to guide the fuzzing process. For this they use a learning-based strategy, by training a neural network, where symbolic execution is used to extract input features that are likely to drive deeper exploration of smart contract code and then these features are then used to train a classifier, which helps steer the fuzzing process toward more promising input values.

SoliAudit Liao et al. (2019) in 2019, increase also the fuzzing to blockchain parameters like gas and ether explicitly and they use static analysis and fuzzing. By assuming now previous knowledge the first containt a vulnerability analisis based on the complided contract by using a machien learning classifier to then fuzz the contract.

Also in 2019, ContraMasterWang et al. (2022a) focused on enhancing vulnerability detection via a general-purpose oracle. By integrating feedback from control-flow graphs, data-flow graphs and dynamic dictionaries, ContraMaster improved the mutation phase and

Tool	Year	Citation
ReGuard	2018	Liu et al. (2018)
ContractFuzzer	2018	Jiang et al. (2018)
ILF	2019	He et al. (2019)
SoliAudit	2019	Liao et al. (2019)
ContraMaster	2019	Wang et al. (2022a)
EthPloit	2020	Zhang et al. (2020)
GasFuzzer	2020	Ashraf et al. (2020)
sFuzz	2020	Nguyen et al. (2020)
Harvey	2020	Wüstholz and Christakis (2020)
Vultron	2021	[refernce]
Targy	2021	Ji et al. (2021)
Confuzzius	2021	Torres et al. (2021)
Smartian	2021	Choi et al. (2021)
SmartGift	2021	Zhou et al. (2021)
Echidna	2021	[reference]
Beak	2022	Wei Zhang (2022)
xFuzz	2022	Xue et al. (2024)
EtherFuzz	2022	Wang et al. (2022b)
SynTest-S	2022	Olsthoorn (2022)
RLF	2022	Su et al. (2022)
effuzz	2023	Ji et al. (2023)
EF\CF	2023	[reference]
IR-Fuzz	2023	Liu et al. (2023)
ItyFuzz	2023	Shou et al. (2023)
Midas	2024	Chen et al. (2024)
Mau	2024	Chen et al. (2024)
CSAFuzzer	2025	Yang et al. (2025)
TPH-Fuzz	2025	Shi et al. (2025)
VERITE	2025	Kong et al. (2025)

Table 3.1: Identified smart contract fuzzers.

enabled the generation of multi-transaction sequences. Detected behaviors were evaluated by passing the execution trace from an instrumented EVM to the general-purpose oracle.

In 2020, HarveyWüstholz and Christakis (2020), a greybox fuzzer specifically designed for Ethereum smart contracts, was introduced. It addressed two particular challenges: random input mutations and state space exploration. Random input mutations often struggle to reach certain paths, especially when specific input conditions are required. To address this, Harvey introduced "fuzzing with input prediction" (referncia), which does not require any static analysis or constraint solving. Instead, it uses runtime instrumentation—referred to as ‘fcost‘, which calculates the distance from the current execution to the target branch (referencia)—to guide input prediction. When predictions are not viable due to insufficient executions, random mutation is used instead.

To address the challenge of state space exploration—since many bugs require multiple

transactions or function invocations—Harvey introduced "demand-driven sequence fuzzing." Rather than fuzzing functions independently or exhaustively exploring all transaction sequences, this method focuses only on sequences that end with the function containing the target branch. All previous transactions are treated as setup steps that prepare the contract's state. Path identifiers and coverage information are then computed only for the final transaction.

Also in 2020, GasFuzzerAshraf et al. (2020) was introduced, again focusing on Ethereum and targeting gas-related vulnerabilities. Built on top of ContractFuzzerJiang et al. (2018), it incorporated two strategies to enhance the detection of gas-related security vulnerabilities: the gas-greedy strategy, which prioritizes transactions that consume more gas and the gas-leveling strategy, applied to transactions that passed the first strategy, to test whether manipulating gas allowances (making them insufficient) causes unexpected changes to the blockchain state.

Also in 2020, sFuzzNguyen et al. (2020) was introduced, combining strategies from AFL. Based on previously developed oracles to detect vulnerabilities, sFuzz supports 8 oracles inspired by previous works. sFuzz uses libFuzzer to implement the fuzzing strategy, identifying functions that do not change the state by performing static analysis of the contract's ABI. It also incorporates a "runner" component in its architecture to set up the fuzzer. Strongly influenced by AFL, it adapts many AFL algorithms to better suit smart contract fuzzing.

ETHPloitZhang et al. (2020), also from 2020, uses taint analysis to detect dependencies among variables. It then generates test cases based on this analysis and executes them in an instrumented EVM. It collects feedback not only in terms of code coverage, but also function rewards and seed value prioritization, focusing on critical instructions that modify storage or perform external calls.

In 2021, SmartianChoi et al. (2021) appeared, combining static and dynamic analysis with fuzzing. Static analysis is used to construct the initial seed corpus, while dynamic analysis provides data-flow feedback during fuzzing. The motivation behind using data-flow feedback is that many transaction sequences leading to vulnerabilities can be identified by analyzing dependencies between functions and persistent state variables.

Also in 2021, TargyJi et al. (2021) was introduced. It integrates taint analysis to help the fuzzer reach deep paths, where complex constraints in conditional statements may limit coverage. Built on top of sFuzz, specifically its libFuzzer component, Targy adds two steps: first identifying parameters involved in conditionals, and then using this information to improve the mutation strategy.

In 2022, RFL Su et al. (2022) was published, focusing on detecting sophisticated vulnerabilities that require transaction sequences. It uses reinforcement learning to model smart contract fuzzing as a Markov Decision Process, balancing code coverage and vulnerability discovery. It is based on earlier work such as ILF He et al. (2019).

Also in 2022:

1. SynTest-Solidity He et al. (2019) appeared, focusing on usability and scalability from a developer's perspective, though the paper lacks technical detail.

2. EtherFuzz Wang et al. (2022b) targeted transaction-ordering-dependent (TOD) vulnerabilities, but again, lacks depth and detail.

3. Beak Wei Zhang (2022), also introduced in 2022, is another low-detail tool with insufficient implementation insights.

xFuzz Xue et al. (2024), similar to sFuzz, was the first to explicitly target cross-contract vulnerabilities such as multi-contract reentrancy. These scenarios are particularly challenging due to the vastly larger search space. xFuzz Xue et al. (2024) employs machine learning to learn attack patterns, and guides the fuzzing process using function and caller priority scores given by the model to help the fuzzer focus on higher-risk code paths or areas.

In 2023, IR-fuzz Liu et al. (2023) was introduced, based on sFuzzNguyen et al. (2020). It aims to address function dependencies and the difficulties in exploring deep branches without wasting too much time in initial states. It directly tackles these issues by analyzing read and write dependencies, applying distance-based schemas , and prioritizing rare branches.

Also in 2023, Effuzz Ji et al. (2023) was proposed, also based on sFuzz Nguyen et al. (2020). Its goal is to improve code coverage by implementing a selection strategy for input parameters and an accelerated multi-objective search method. First, Effuzz uses taint analysis to obtain the set of parameters involved in conditional statements. Then, it improves the mutation strategy over these parameters to reduce or eliminate invalid mutations. The mutation strategy is based on two criteria:

1. Reachability: Ensuring that mutated inputs can reach the conditional branch under its constraints.

2. Satisfiability: Ensuring that parameters irrelevant to the conditional are not mutated, thereby reducing mutation overhead.

ItyFuzz Shou et al. (2023), introduced in 2023, is a snapshot-based fuzzing framework for Ethereum smart contracts that achieves efficient state-space exploration by avoiding repeated re-execution of long transaction sequences. Its architecture builds on the libAFL fuzzing engine and leverages the FuzzFactory approach of using intermediate "waypoint" feedback mechanisms. In addition to standard coverage-guided feedback, ItyFuzz incorporates multiple domain-specific waypoints: a dataflow waypoint mechanism that guides exploration toward promising contract states, and a comparison waypoint mechanism that prunes less-interesting state branches. It also includes the possibility to fuzz "on - chain ".

Midas Ye et al. (2024) appears, based on ItyFuzz, focusing of profitable vulnerabilities, for this they introduce two new feedback mechanisms. The first one Validity Waypoint: Hooks jump-related opcodes to identify new jump destinations and discards reverted seed

inputs the second one: Performance Waypoint: Hooks call-related opcodes to track token flows and identify seed inputs with new token transfer behaviors o track diverse token flows to efficiently explore the state space of smart contracts.

Verite Kong et al. (2025) appears, persuading the finding of maximum profitability when a vulnerability that obtains profitability is detected they achieve this by primarly use gradient descent-based profit maximization strategy for these identified candidates based on a model of action based instead of API level that models the actions as a group of transactions or API level calls

3.3.2 Challenges and Research Gaps in Ethereum-Specific Fuzzing

Fuzzing Ethereum smart contracts inherits the general difficulties of fuzzing while introducing a set of unique challenges becouse of blockchain execution environment, contract semantics, and decentralized application (dApp) ecosystems. Despite notable advances in tools such as ContractFuzzer, ReGuard, or ItyFuzz, important obstacles remain. Below we group the main challenges and research gaps:

Inter-Contract Vulnerabilities. A major limitation of current Ethereum fuzzers is their predominant focus on single-contract analysis, even the fact that many real-world exploits involve multiple interacting contracts. Attacks involving flash loans, oracle manipulation, or cross-DEX arbitrage typically require carefully crafted call chains across several contracts, something beyond the scope of existing fuzzers.

Bug Detection and Oracle Design. Ethereum fuzzing faces amplified challenges in oracle quality because most vulnerabilities are logical, economic, or semantic rather than crash-inducing. For example, improper authorization, reentrancy, or transaction-ordering dependence (TOD) may leave a contract vulnerable without ever triggering a crash. Tools like ReGuard, EtherFuzz, and GasFuzz encode oracles for specific bug classes, while systems like Midas and Verite explicitly target exploitability and profitability. However, there is still no broadly applicable oracle framework that can detect a wide range of semantic bugs with high precision. Moreover, benchmarking oracle effectiveness is also a need. Developing sophisticated, general-purpose oracles that can reason about semantic properties (e.g., invariant violations, misaligned balances, unexpected fund transfers) remains a central research gap.

State-Space Explosion and Transaction Sequences. The persistent state of Ethereum contracts leads to a combinatorial explosion of possible execution states. Vulnerabilities often require long sequences of dependent function calls, which are hard to discover with standard fuzzing heuristics. Although tools such ItyFuzz introduces snapshot-based fuzzing to revisit promising states, exhaustive exploration remains computationally infeasible. The research

challenge lies in defining scalable abstractions or guidance mechanisms that allow fuzzers to prioritize semantically meaningful paths without being overwhelmed by the exponential state space.

Environment-Specific Constraints. The Ethereum execution environment imposes additional obstacles not present in traditional software fuzzing. for instance, gas limits, timestamps, block mining behavior, storage costs, are critical for realism but challenging to implement. Simplified or incomplete models risk both false negatives and false positives. Ensuring semantic fidelity to real on-chain behavior remains an open problem.

Benchmarking, Reproducibility, and Standards. Finally, Ethereum fuzzing research suffers from a lack of standardized benchmarks, evaluation metrics, and reproducibility practices. Tools differ in the datasets they use (synthetic contracts, real-world DeFi systems, or exploited vulnerabilities), their time budgets, and their stopping criteria, making comparisons unreliable. No accepted benchmark suite exists that combines vulnerable contracts, diverse real-world examples, standardized budgets, and metrics. Similarly, evaluation standards (e.g., reporting precision/recall of oracles, time-to-first-exploit, max profit extraction) remain inconsistent. The lack of such a unifying framework make difficult cumulative progress in the field.

In conclusion, while Ethereum fuzzing has achieved notable progress and remains the most developed area within smart contract testing, several challenges and research gaps persist. Addressing these issues is crucial to move Ethereum fuzzing beyond a collection of early-stage prototypes toward a more mature, robust, and consolidated methodology.

Chapter 4

Design and Implementation: An Experimental Perspective

One of the fundamental challenges in fuzzing, and particularly in smart contract fuzzing, is the state space explosion problem. This phenomenon is not only present in fuzzing but it has been recognized also in symbolic execution or formal verification. While traditional software fuzzing has acknowledged and partially addressed this issue through corpus management and related techniques, systematic solutions in the context of smart contracts remain very few. Among the few existing efforts, ItyFuzz Shou et al. (2023) represents one of the first frameworks to approach the problem in a structured manner.

The present work addresses the state space explosion problem in smart contract fuzzing by introducing a strategies based in streaming and probabilistic techniques. Central to this approach is the application of *streaming distinct element estimation* Chakraborty et al. (2023), which provides a mechanism for approximating the number of unique states explored during fuzzing without requiring exhaustive storage. When applied in this context we try to maintain a representative subset of a contract's reachable states while simultaneously allowing the reconstruction of the global state distribution with statistical guarantees. This allows to avoid the prohibitive costs associated with full state enumeration and unbounded computation.

The proposed approach is developed under the assumption that no metric exists which can *a priori* identify states that are more likely to lead to vulnerabilities. Were such a metric available, it would imply prior knowledge of the vulnerabilities themselves. Common heuristics such as code coverage, while effective for guiding exploration, have not demonstrated a reliable correlation with vulnerability discovery . For this reason, the methodology adopted in this thesis emphasizes probabilistic techniques designed to balance state-space diversity with resource usage.

4.1 Strategies Design

To address the problem of uncontrolled state growth during fuzzing, we designed and implemented pruning strategies extendeding the one proposed by ItyFuzz. As the fuzzer continuously generates states snapshots of Ethereum smart contracts, the corpus of stored states expands rapidly. Without pruning, this growth leads to excessive memory usage and reduced efficiency, as many states provide little additional value. Pruning is therefore essential to control corpus size while preserving sufficient diversity to maximize coverage and bug discovery.

This chapter introduces three strategies for corpus pruning. The first, the *Voting Strategy*, is inherited from the original ItyFuzz. The second, F_0 *Pruning*, applies a probabilistic technique derived from distinct element estimation Chakraborty et al. (2023) to maintain a representative corpus with statistical guarantees. Finally, F_0 *Voting* combines deterministic prioritization with probabilistic pruning to balance systematic selection and randomness.

4.1.1 Voting Strategy

The baseline Voting Strategy originates from the design of the original ItyFuzz and addresses the need to eliminate states when the state corpus becomes too large. The key idea is to assign each state an "interestingness" score in the form of votes, and track how often each state has been visited (executed). If the corpus reaches a predetermined size threshold, the fuzzer will remove the states that have proven least interesting relative to how many times they were tried.

In ItyFuzz context, a "vote" is granted to a state whenever an execution from that state discovers new information or coverage. For example, ItyFuzz uses a comparison waypoint feedback: if executing a state yields a new minimum in some comparison metric (i.e. getting two values closer than ever before for a branch condition), that state earns an extra vote. In this way each vote encodes that the state was interesting enough to contribute something new to exploration.

This voting mechanism is used together with a visit count for each state, recording how many times the fuzzer has selected that state for execution. Together, these two counters allow the strategy to compute a simple performance ratio: votes/visits. A low votes/visits ratio means a state has been tried many times without much benefit. The Voting Strategy prunes precisely those states once the corpus size limit is exceeded. By doing so, it removes the least interesting but most explored states preventing the corpus from becoming saturated with unsuccessful states

4.1.2 F_0 Pruning

The F_0 *Pruning* strategy is inspired by algorithms from the data-stream literature, in particular the distinct-elements (F_0) problem. In a streaming context, given a stream $A = \langle a_1, a_2, \ldots, a_m \rangle$, the quantity $F_0(A)$ denotes the number of distinct elements. introduced an efficient algorithm for approximating F_0 without storing the entire stream, instead maintaining a bounded random sample with dynamically adjusted sampling probability.

Within the F_0 *Pruning* strategy, the estimator plays a dual role: it determines both when pruning should occur and which states are most eligible for removal. Pruning is triggered whenever the estimator distinct-count approximation exceeds the configured maximum number to states in the state corpus, signaling that the corpus has grown beyond the intended bound. The estimator internal sample set X then acts as a diversity-preserving core of the corpus: states in X are retained, while pruning is applied only to states outside of it. In typical cases, this provides a sufficiently large candidate pool to remove exactly the number of states to be pruned. If the candidate pool is too small, the strategy falls back to the baseline heuristic of ranking states by their vote-to-visit ratio, ensuring that the required number of removals is always met. In this way, pruning decisions are probabilistically guided by the estimator, with deterministic ranking used only as a safety net.

This design tries to guarantee that pruning events that are both efficient and diversity-preserving. In the early stages, when many redundant states are produced, pruning aggressively eliminates states outside the sample. As fuzzing progresses, the down sampling ensures that X remains bounded but continues to represent a random set of the state space. By relying on probabilistic sampling rather than deterministic heuristics, F_0 Pruning avoids consistently discarding the same types of states and reduces the risk of prematurely eliminating states that may later be valuable.

```
1  Parameters: DROP_THRESHOLD, PRUNE_AMT
2  F0_EPSILON, F0_DELTA, STREAM_MULTIPLIER
3
4  Estimator State:
5    X :=                          // sampled state indices
6    p := 1.0                     // current sampling probability
7    thresh := ceil((12 /   ^2) * ln(8 * (DROP_THRESHOLD *
       STREAM_MULTIPLIER) /   ))
8
9  Estimator.add(i):
10   if i in X: remove i
11   if random() < p: X.insert(i)
12   if |X| >= thresh:
13     X := { x in X | random_bool() }   // keep each with prob 1/2
14     p := p / 2
```

```
15
16 Scheduler.on_add(new_idx):
17   add new_idx to corpus (votes=V0, visits=1)
18   Estimator.add(new_idx)
19
20   if Estimator.estimate() > DROP_THRESHOLD:
21     candidates := { i     corpus | i     X and i     3 and i
      new_idx }
22     to_remove := []
23
24     if |candidates|     PRUNE_AMT:
25       to_remove := first PRUNE_AMT from candidates
26     else:
27       ranked := states sorted by votes/visits ascending
28       for s in ranked:
29         if s     X and s     3 and s     new_idx:
30           to_remove.append(s)
31           if |to_remove| == PRUNE_AMT: break
32
33     for each s in to_remove:
34       corpus.remove(s)
```

Listing 4.1: F_0 Pruning Strategy. States are sampled into a bounded set X with adaptive probability p. Pruning removes states outside X, with a fallback to score-based removal if too few candidates exist.

4.1.3 F_0 Voting

The F_0 *Voting* strategy represents a hybrid approach that combines the deterministic ranking mechanism of the baseline Voting strategy with probabilistic pruning and a mechanism to gradually reduce pruning intensity over time. Like the other strategies, pruning is triggered whenever the size of the state corpus exceeds a fixed threshold. However, instead of deterministically removing the worst-ranked states, F_0 Voting introduces controlled randomness into the pruning process and dynamically adjusts its aggressiveness as fuzzing progresses.

When pruning is triggered, the states are ranked in ascending order according to their vote-to-visit ratio, as in the baseline strategy. Rather than removing the lowest-ranked states outright, each candidate for removal is discarded only with probability p, referred to as the current prune probability. In practice, this corresponds to iterating over the least promising states and, for each,"flipping a coin" with probability p to determine whether it should be removed. On average, this procedure eliminates $p\times$ "defined number of states to be pruned" states per pruning event, but the actual subset removed varies, thereby preventing overly

rigid behavior. If $p = 1.0$, the strategy reduces to the deterministic baseline; if $p = 0.5$, approximately half of the "defined number of states to be prunned" candidates are removed at random.

Following each pruning event, the prune probability is updated according to a decay factor $d < 1$, such that p is multiplied by d. This ensures that pruning begins aggressively but becomes increasingly conservative over time. For example, with an initial $p = 1.0$ and $d = 0.9$, the prune probability decreases geometrically $(1.0, 0.9, 0.81, \dots)$, eventually approaching zero. Consequently, in the early stages of fuzzing, when many generated states are repetitive, the strategy aggressively prunes these low-value states. As the campaign progresses and deeper, more complex states are discovered, pruning slows down, thereby reducing the risk of discarding valuable states. The idea behind this strategy tries to reflect that in fuzzing early phases teh fuzzer generates abundant trivial states, while later phases yield fewer but more meaningful states.

The pseudocode for the F_0 Voting strategy is shown in Algorithm 4.2. It illustrates how states are selected probabilistically for removal, and how the pruning probability decays after each event to gradually reduce pruning probability.

```
1  Parameters: DROP_THRESHOLD, PRUNE_AMT
2  INITIAL_PROB (initial prune probability, e.g. 1.0)
3  PROB_DECAY (decay factor per prune, e.g. 0.9)
4  State: current_prob (current prune probability, init = INITIAL_PROB)
5
6  on add_new_state(new_state):
7      add new_state to corpus (votes=V0, visits=1)
8      if corpus.size > DROP_THRESHOLD:
9          sorted_states = sort(corpus.states, by=votes/visits ascending
       )
10         to_remove = []
11         count = 0
12         for each s in sorted_states:
13             if s is not seed and s != new_state:
14                 if random() < current_prob:
15                     to_remove.append(s)
16                     count += 1
17                     if count == PRUNE_AMT:
18                         break
19         // Decay the probability for next time
20         current_prob = current_prob * PROB_DECAY
21         // Remove the chosen states
22         for each s in to_remove:
```

```
23                 corpus.remove(s)
```

Listing 4.2: F_0 Voting Strategy. This probabilistic algorithm removes low-ranked states with a certain probability, and reduces that probability after each pruning, thereby lessening pruning aggressiveness over time.

4.2 Strategies Experiment Implementation

We implemented our strategies on top of ItyFuzz, a state-of-the-art hybrid fuzzer for smart contracts. ItyFuzz was chosen for several reasons:

- **Snapshot-Based Design:** ItyFuzz introduces a novel snapshot-based fuzzing approach . Instead of treating a sequence of transactions as a monolithic input, it stores *snapshots* of intermediate states (VM states) in a corpus. This aligns perfectly with our need to manage a corpus of states. The concept of snapshotting gives a natural point at which to apply our pruning strategies.

- **Performance and Extensibility:** ItyFuzz provides a modular and well structured architecture that facilitates the incorporation of new strategies.This extensibility enables the integration of custom corpus schedulers and state management mechanisms without requiring the design of a new fuzzer.

Concretely, we integrated our strategies by extending ItyFuzz, in particular the corpus scheduler and state logic for adding new inputs (states) including our pruning triggers.

All our modifications are isolated behind feature flags (e.g., features *f0_pruning* or *f0_voting*), so one can run ItyFuzz in original mode or with each strategy turned on, for fair comparisons.It is also important to mention that we did not need to alter how inputs are generated or executed.

4.3 Experimental Environment and Reproducibility

Our experiments were conducted in a controlled environment to ensure reliable results and facilitate reproducibility. All fuzzing experiments were performed on an AlmaLinux 9.5 machine with an Intel i7-6700HQ CPU (4 cores, 8 threads) and 8 GB of RAM.

We used Rust 1.70+ to compile ItyFuzz with our modifications. The code was built in `-release` mode for maximum performance . For runs using our strategies, we enabled the corresponding feature flags at compile time (e.g., `cargo build -release -features f0_pruning`).

We took care to log all relevant configuration and parameters at the start of each run. Our tool prints out the values of key settings (e.g., DROP_THRESHOLD, PRUNE_AMT,

ϵ, δ, initial prune probability) on startup. This is included in the fuzzer's output to aid in verifying how the fuzzer was configured for a given experimental trial.

Our implementation allows tuning of the various parameters discussed, either through command-line arguments or compiled constants. We created an initialization routine for each set of parameters which is invoked at program start (or when the feature is enabled). Logging statements in these routines print out the configured values, which helps in verifying experimental settings.

4.3.1 Benchmark Suite

To evaluate our pruning strategies, we used a focused set of motivating smart contracts drawn from prior work, together with one synthetic contract crafted for this thesis to exaggerate the state-space explosion problem. All Solidity sources are provided in Appendix A for reproducibility.

Table 4.1: Benchmark smart contracts used in evaluation. All code is included in Appendix A.

Contract	**Motivation Example of**
`SimpleState`	ItyFuzz
`Crowdsale`	IFL
`BazExample`	Harvey
`ComplexState`	Synthetic (this work)

Each fuzzing run was conducted with a fixed time of 30 seconds per contract. For every contract, we executed all three pruning strategies in parallel:Voting Strategy, F_0 Voting and F_0 Pruning. All three strategies were compiled separately and executed on the same contracts under identical conditions. This design ensures that differences in outcomes can be attributed directly to the pruning logic rather than to external factors such as environment setup.

The experiment sets vary several parameters for each strategy. Two parameters are shared by all strategies: the *drop threshold* (maximum allowed corpus size before pruning is triggered) and the *prune amount* (number of states removed during each pruning event). F_0 Voting extends this baseline with two additional parameters: the *initial pruning probability* (probability of removing a candidate state at the beginning of fuzzing) and the *decay factor* (rate at which this probability is reduced after each pruning round). F_0 Pruning instead relies on three parameters specific to its streaming distinct-element estimator: the *error tolerance* ϵ, the *failure probability* δ, and the *stream-length multiplier*, which scales the estimator internal window size.

Table 4.2 summarizes all experiment sets and their parameters. Each row corresponds to one configuration, which was applied consistently across all three pruning strategies. For Voting, only *Drop* and *Prune* are relevant; for F_0 Voting, the additional *Init Prob.* and

Decay columns are used; and for F_0 Pruning, the estimator parameters ϵ, δ, and *Multiplier* are used.

Table 4.2: Experiment sets with parameters for all three pruning strategies. Shared parameters (*Drop*, *Prune*) apply to all strategies; the remaining columns apply only to F_0 Voting or F_0 Pruning.

Set	**Drop**	**Prune**	**Init Prob.**	**Decay**	ϵ	δ	**Multiplier**
baseline	100	25	1.0	0.5	0.08	0.02	2
thresh_low	50	25	1.0	0.5	0.08	0.02	2
thresh_high	150	25	1.0	0.5	0.08	0.02	2
amount_low	100	15	1.0	0.5	0.08	0.02	2
amount_high	100	35	1.0	0.5	0.08	0.02	2
prob_conserv.	80	20	0.3	0.5	0.08	0.02	2
prob_aggress.	80	20	1.0	0.5	0.08	0.02	2
decay_slow	80	20	0.8	0.9	0.08	0.02	2
decay_fast	80	20	0.8	0.2	0.08	0.02	2
precision_high	90	30	1.0	0.5	0.05	0.01	2
precision_low	90	30	1.0	0.5	0.15	0.05	2
memory_short	90	30	1.0	0.5	0.08	0.02	1
memory_long	90	30	1.0	0.5	0.08	0.02	4
aggressive	30	25	0.9	0.3	0.20	0.10	1
conservative	200	10	0.5	0.8	0.03	0.005	4

In summary, the benchmark suite presented in Table 4.1 provides a representative set of contracts with varying characteristics. This diversity is essential for evaluating the performance of the proposed fuzzing strategies under different conditions and for ensuring that the results are not biased toward a single contract type. While the parameters in 4.2 define the experimenal settings. Together, they provide the basis for the empirical evaluation presented in the following section.

Chapter 5

Experimental results and discussion

We now present the results of our experiments and discuss how in practice on our smart-contract benchmarks. Our evaluation focuses on three central aspects: (i) fuzzing efficiency (coverage), (ii) vulnerability discovery, and (iii) resource usage.

Fuzzing efficiency (coverage) When comparing the three strategies in terms of code coverage (instruction and branch coverage), both across different parameterizations and benchmarks, we observe only limited differences (Figure 5.1). This suggests that all three strategies are capable of reaching similar levels of coverage in equal time.

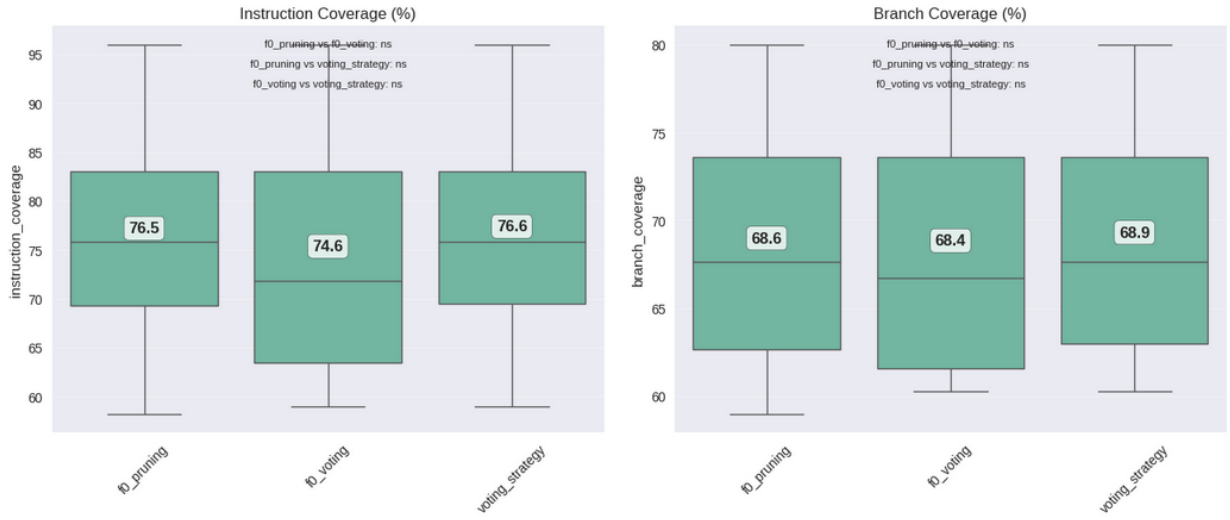

Figure 5.1: Instruction and Branch Coverage for the three strategies

However, when examining the results at the level of individual benchmarks, some differences can be appreciated. In particular:

On the IFL contract, F0_Pruning achieved lower coverage than both the baseline Voting strategy and F0_Voting. This indicates that the diversity-aware pruning heuristic, while generally effective, may occasionally be too aggressive and discard states that are useful for deeper exploration in specific contract structures (Figure 5.2).

On the synthetic benchmark (the stress-test created for this work to maximize state explosion), the weakest performance came from F0_Voting, which reached a lower coverage level than the other two strategies (Figure 5.3) . This could reflect the fact that probabilistic pruning can sometimes delay exploration of deeper or rarer states, especially under stress conditions.

Overall, across the other benchmarks, the three approaches performed comparably in terms of final coverage.

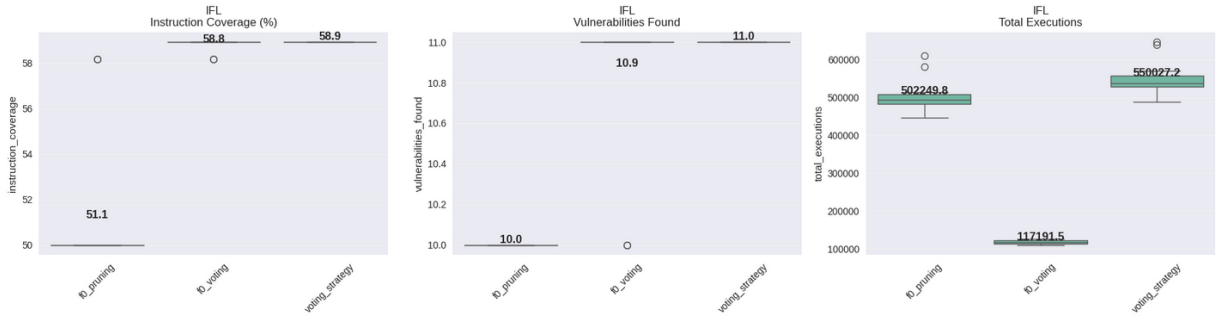

Figure 5.2: Instruction coverage, vulnerabilities found and total executions for the three strategies in IFL motivation example

Vulnerability discovery The trends in vulnerability discovery mirror those of coverage. All three strategies discovered the critical vulnerabilities embedded in the benchmarks, but their relative effectiveness varied depending on the target:

On IFL, F0_Pruning lagged behind the Voting Strategy and F0_Voting in terms of the number of distinct vulnerabilities triggered (Figure 5.2).

On the synthetic benchmark, F0_Voting underperformed, while F0_Pruning achieved results comparable to the baseline (Figure 5.3).

In general, the strategies converged on finding the same vulnerabilities eventually, but the number of executions required to reach them differed. This indicates that pruning strategy primarily affects efficiency rather than effectiveness: the same bugs are found, but with different resource costs.

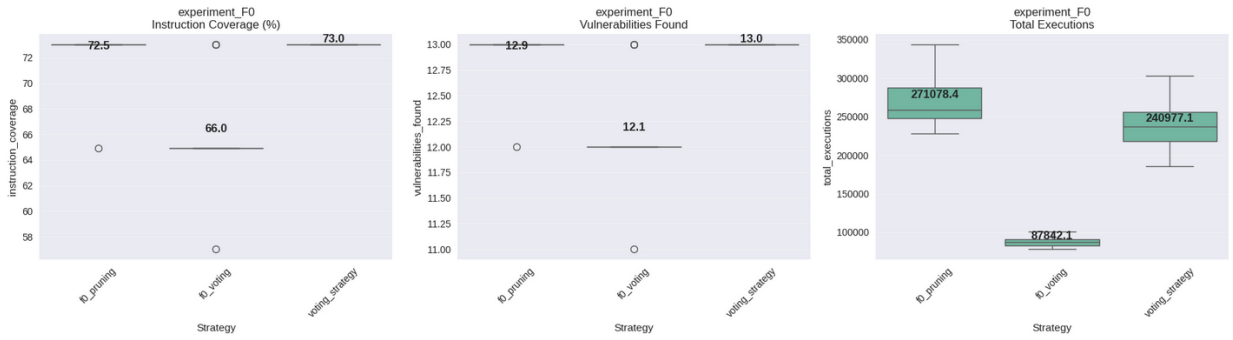

Figure 5.3: instruction coverage, vulnerabilities found and total executions for the three strategies in synthetic case

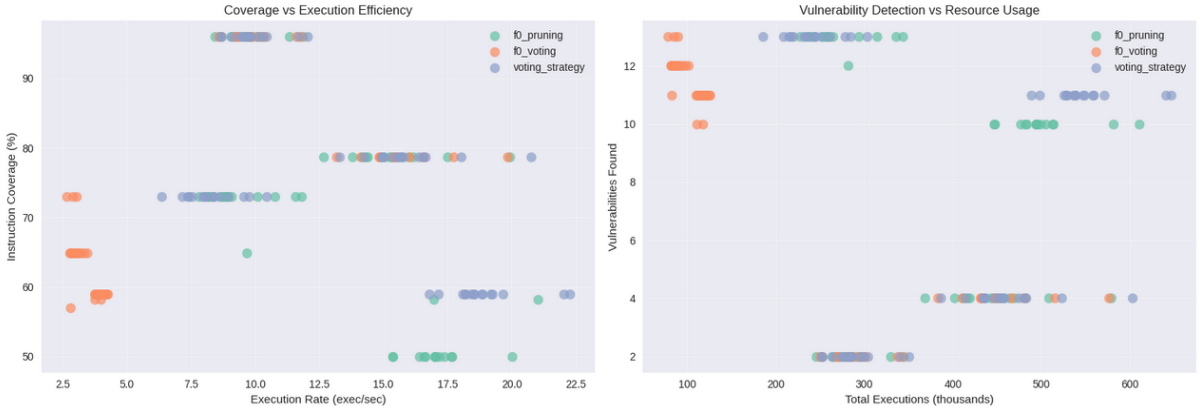

Figure 5.4: Comparison between strategies

Resource usage For what regards, resource utilization F0_Voting executed substantially fewer test cases than the other two strategies in almost all benchmarks.

Interestingly, while F0_Voting was less efficient in terms of executions per second, it still achieved similar coverage and vulnerability discovery to the other strategies. This suggests that it may make better use of each execution.

Overall discussion. From these results, we draw several conclusions:

- Strategy performance depends on the target. On IFL, F0_Pruning lagged behind, while on the synthetic benchmark, F0_Voting did. This suggests that no strategy dominates across all workloads, and their effectiveness is influenced by contract structure and the nature of the state space.
- F0_Voting is more efficient per execution. Even executing fewer test cases, it reaches similar coverage and bug counts, highlighting that its probabilistic pruning can sometimes achieve good results.
- Voting strategy remains competitive. It provides stable performance, but without explicit consideration of state diversity it may spend effort on redundant states.

The current experiments were conducted on a limited benchmark suite and with relatively short time budgets (30 seconds per run). To fully validate the observed trends, further evaluation is needed on:

- Larger and more diverse contract sets, including real-world contracts with complex interactions.
- Longer fuzzing runs, to assess whether the probabilistic strategies (particularly F0_Voting) reveal delayed advantages.

- Isolated analysis of the underlying algorithms (the distinct-elements estimator and the probabilistic pruning rule) on larger data streams, to measure their accuracy and robustness without being influenced by the Voting strategy.

These extensions would help clarify under what conditions each strategy is superior, and provide guidance on how to tune them for different fuzzing workloads.

Chapter 6

Conclusions and Future Work

This thesis examined fuzz testing in the context of Ethereum smart contracts, addressing both the general challenges of fuzzing and those specific to blockchain environments. Through a literature review and a targeted experimental study, we answered the research questions and obtained a clearer understanding not only of fuzzing's current capabilities and limitations but also of the directions future research should take.

With respect to fundamental challenges in fuzz testing (RQ1), our findings reaffirm that despite decades of progress, fuzzing still lacks rigorous theoretical foundations. We have no reliable models to estimate the proportion of bugs a campaign can expose or the point at which returns diminish. Similarly, widely used metrics such as code coverage or execution speed are inadequate proxies for security assurance, and no standardized benchmarks exist to allow meaningful comparison of techniques. General-purpose fuzzers also struggle with structured inputs and complex program states, and bug detection remains largely limited to crashes, leaving logical and semantic vulnerabilities beyond reach.

These issues are amplified in the case of smart contracts (RQ2). Contracts operate in persistent and stateful environments where vulnerabilities often emerge only through particular transaction sequences or inter-contract interactions, leading to state-space explosion. Current fuzzers largely focus on single contracts and single transactions, thereby missing cross-contract vulnerabilities common in real exploits. Moreover, the oracle problem is even more important, since most smart contract failures are economic or logical rather than crashes. In practice, fuzzing smart contracts inherits the difficulties of general fuzzing while adding new domain-specific ones. Given that today "fuzzing smart contracts" is almost synonymous with fuzzing Ethereum, the observations for RQ2 and RQ3 converge. These results also directly address SRQ1, showing that existing methodologies only partially capture the particularities of EVM-compatible blockchains

Our study also considered whether techniques from general software fuzzing could benefit smart contracts, and vice versa (RQ4). Structure-aware and guided methods from traditional fuzzing remain underutilized in this domain, while conversely, innovations such as snapshot-based fuzzing and oracles from the smart contract can benefit general software fuzzing.

Realizing this cross benefiting remains an important research topic. Beyond Ethereum, fuzzing is still in its infancy. While Ethereum dominates both in usage and research maturity, platforms such as Solana or Move-based blockchains have only anecdotal support for fuzz testing. This immaturity reflects both Ethereum's dominance in adoption and the technical divergences that prevent direct transfer of tools, underscoring the need for platform-specific approaches. This finding corresponds to SRQ2, confirming that fuzzing methodologies for non-EVM blockchains remain at an early and immature stage.

Within this context, the thesis makes both theoretical and practical contributions. Theoretically, it systematized the state of the art, categorizing limitations and highlighting critical areas for further work, thereby directly addressing RQ1–RQ4 and the secondary questions. Practically, it extended the ItyFuzz framework with three strategies designed to mitigate state-space explosion. Our evaluation showed that while all three strategies eventually achieved comparable coverage and bug discovery, their behaviors differed depending on the contract. No single strategy dominated, but the experiments suggest that pruning and probabilistic methods can control redundant states without undermining effectiveness. These insights directly inform the conclusion that state-space explosion is manageable but lacks a single solution.

The findings suggest several directions for further research. First, fuzzing requires stronger theoretical models and standardized benchmarks. Models capable of quantifying bug discovery and stop criteria would allow fuzzers to allocate resources more effectively, while benchmark suites with diverse contracts and vulnerabilities would enable reproducible and comparable evaluation. Second, there is also a need for advanced oracles that can detect logical vulnerabilities beyond crashes. Third, fuzzing must move beyond single-contract execution to multi-contract and multi-transaction scenarios, and must adapt to non-EVM platforms where new execution models demand novel approaches. Finally, improving efficiency and scalability remains essential. Our results with probabilistic pruning illustrate the potential of algorithms from other fields, future work could extend this with adaptive strategies, GPU acceleration, or distributed fuzzing integrated into continuous testing pipelines.

Bibliography

Nick Szabo. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, 18(16), 1996.

Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Yellow paper, Ethereum Project, 2014. URL `https://ethereum.github.io/yellowpaper/paper.pdf`.

Muhammad Mehar, Charlie Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Journal of Cases on Information Technology*, 21(1):19–32, 2017. doi: 10.2139/ssrn.3014782.

Rekt-News. Bybit – rekt: $1.43b heist on bybit claims the throne on our rekt leaderboard, February 2025.

Vitalik Buterin. Possible futures of the ethereum protocol, part 6: The splurge. Blog post on Vitalik's personal site, October 2024. URL `https://vitalik.eth.limo/general/2024/10/29/futures6.html`.

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186. Springer. ISBN 978-3-662-54455-6. doi: 10.1007/978-3-662-54455-6_8.

Ethereum Foundation. Trillion dollar security: Security challenges overview. Technical report, Ethereum Foundation, June 2025. URL `https://ethereum.org/en/trillion-dollar-security/`. First deliverable of the Trillion Dollar Security (1TS) project.

Areta Research. The state of crypto security 2025. Technical report, Areta Research, July 2025. URL `https://research.areta.io/state-of-crypto-security-2025`. Accessed via Areta Research site.

Blockchain Security Standards Council, Inc. General security and privacy guidelines (gsp). Technical report, Blockchain Security Standards Council, Inc., June 2025. URL

https://specs.blockchainssc.org/gsp/. Baseline risk management, security, and privacy guidelines; aligned with NIST-CSF.

Peiyu Wang and CertiK Team. Web2 meets web3: Hacking decentralized applications. Technical report, CertiK, August 2024. URL https://www.certik.com/resources/blog/web2-meets-web3-hacking-decentralized-applications. Originally presented at AppSec Village, DEF CON 32 on August 10, 2024.

Mujtaba Raza. The hidden threats of web2 vulnerabilities in web3 systems. Technical report, BlockApex, May 2025. URL https://blockapex.io/the-hidden-threats-of-web2-vulnerabilities-in-web3-systems/.

Blockchain Security Standards Council. Blockchain security standards. Technical report, Blockchain Security Standards Council, Inc., June 2025. URL https://www.blockchainssc.org/standards. Includes NOS, TIS, KMS, and General Security and Privacy Guidelines.

David Wong and Mason Hemmel. Dasp: The decentralized application security project. Technical report, NCC Group / CryptoServices, 2018. URL https://www.dasp.co/. Includes the DASP Top-10 vulnerabilities list for smart contracts.

Consensys Diligence. Swc registry: Smart contract weakness classification and test cases. Technical report, Consensys Diligence, 2019. URL https://swcregistry.io/.

Zhe Wei, Jing Sun, Zhi Zhang, Xin Zhang, Xiaoyu Yang, and Lei Zhu. Survey on quality assurance of smart contracts. *ACM Computing Surveys*, 57(2):1–36, 2025. doi: 10.1145/3695864.

OWASP Foundation. Smart contract security verification standard (scsvs). Technical report, OWASP Foundation, 2025. URL https://owasp.org/www-project-smart-contract-security-verification-standard/.

Enterprise Ethereum Alliance. Eea ethtrust security levels specification, version 2. Technical report, Enterprise Ethereum Alliance, Inc., December 2023. URL https://entethalliance.org/specs/ethtrust-sl/v2/.

Haozhe Zhou, Amin Milani Fard, and Adetokunbo Makanju. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *Journal of Cybersecurity and Privacy*, 2(2):358–378, may 2022. doi: 10.3390/jcp2020019. URL https://doi.org/10.3390/jcp2020019.

Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, 2023. IEEE. doi: 10.1109/ICSE48619.2023.00061.

Nethermind Research. Eip-7702 attack surfaces: What developers should know, 2025. URL https://www.nethermind.io/blog/eip-7702-attack-surfaces-what-developers-should-know.

Three Sigma. How eip-7702 transforms account security and functionality, May 2025. URL https://x.com/threesigmaxyz/status/1924782348601872878.

Mounira Kezadri Hamiaz and Maha Driss. Ethereum smart contracts under scrutiny: A survey of security verification tools, techniques, and challenges. *Computers*, 14 (6):226, 2025. doi: 10.3390/computers14060226. URL https://doi.org/10.3390/computers14060226.

Bo Jiang, Ye Liu, and W.K. Chan. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269, September 2018. doi: 10.1145/3238147.3238177. URL https://ieeexplore.ieee.org/document/9000089. ISSN: 2643-1572.

Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys*, 54(7):1–38, July 2021. doi: 10.1145/3464421. URL https://doi.org/10.1145/3464421.

Certora Ltd. Certora prover, August 2025. URL https://github.com/Certora/CertoraProver.

argotorg. hevm: Symbolic and concrete evm execution engine, 2025. URL https://github.com/argotorg/hevm.

Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Confusum contractum: confused deputy vulnerabilities in ethereum smart contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1793–1810, 2023.

Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Not your type! detecting storage collision vulnerabilities in ethereum smart contracts. In *Network and Distributed Systems Security (NDSS) Symposium 2024*, 2024.

Rekt.News. Cork protocol – rekt: Fake tokens just popped the cork protocol for $12 million, May 2025. URL https://rekt.news/cork-protocol-rekt.

Cork Protocol Docs. *Audits – Cork Protocol Documentation*, 2025. URL https://docs.cork.tech/smart-contracts/v1/audits.

nican0r. The bug that was missed: How fuzzing for preconditions can lead to high severity vulnerabilities, May 2025. URL https://getrecon.substack.com/p/the-bug-that-was-missed.

Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities, December 1990. URL https://doi.org/10.1145/96267.96279. Communications of the ACM, Vol. 33, No. 12.

Oulu University Secure Programming Group (OUSPG). Protos – security testing of protocol implementations. http://www.ee.oulu.fi/research/ouspg/protos/, 1999–2001.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. June 2005. doi: 10.1145/1064978.1065036. URL https://doi.org/10.1145/1064978.1065036. ACM SIGPLAN Notices, Vol. 40, No. 6. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05).

Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft., January 2012a. URL https://doi.org/10.1145/2090147.2094081. ACM Queue, Vol. 10, No. 1.

Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: whitebox fuzzing for security testing, March 2012b. URL https://doi.org/10.1145/2093548.2093564. Communications of the ACM, Vol. 55, No. 3, pp. 40–44.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, December 2008. URL https://dl.acm.org/doi/10.5555/1855741.1855756. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*, pp. 209–224. USENIX Association.

Michał Zalewski. American fuzzy lop (afl) fuzzer, 2013. URL http://lcamtuf.coredump.cx/afl/.

Kostya Serebryany. libFuzzer: a library for in-process, coverage-guided, evolutionary fuzzing, 2015. URL https://llvm.org/docs/LibFuzzer.html.

Robert Świecki. honggfuzz: security oriented, general-purpose, feedback-driven fuzzer, 2010. URL https://honggfuzz.dev/.

Google Open Source Security Team. OSS-Fuzz: continuous fuzzing for open source software, 2017. URL https://github.com/google/oss-fuzz.

Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, Jonathan Metzman, and ClusterFuzz team. Open sourcing clusterfuzz: scalable fuzzing infrastructure for continuous security testing, February 2019. URL https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html.

DARPA. Cyber grand challenge (cgc): autonomous cyber-defense competition, 2016a. URL https://www.darpa.mil/research/programs/cyber-grand-challenge.

DARPA. Mayhem declared preliminary winner of historic cyber grand challenge, August 2016b. URL https://www.darpa.mil/news/2016/mayhem-winner-cyber-grand-challenge.

David Brumley. The story of mayhem: The next-generation in application security, May 2021. URL https://www.mayhem.security/blog/the-story-of-mayhem-the-next-generation-in-application-security.

Andrea Fioraldi, Dominik Maier, Marc "van Hauser" Heuse, and Heiko "hexcoder-" Eissfeldt. Afl++: community-maintained fork of american fuzzy lop with enhanced performance and features, 2020. URL https://github.com/AFLplusplus/AFLplusplus.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017a. doi: 10.1109/TSE.2017.2725227.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, 2017b. doi: 10.1145/3133956.3134020.

V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE'19, pages 748–758, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00083. URL https://doi.org/10.1109/ICSE.2019.00083.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):174:1–174:29, oct 2019. doi: 10.1145/3360600. URL https://doi.org/10.1145/3360600.

Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2018. IEEE. doi: 10.1109/SP.2018.00046. URL https://doi.org/10.1109/SP.2018.00046.

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Network*

and Distributed System Security Symposium (NDSS 2017), San Diego, CA, USA, February 2017. Internet Society. URL `https://www.ndss-symposium.org/`. NDSS Symposium 2017.

Google. syzkaller: Unsupervised coverage-guided kernel fuzzer. `https://github.com/google/syzkaller`.

Dave Jones. Trinity: Linux system call fuzzer. `https://github.com/kernelslacker/trinity`.

Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2025. Internet Society. URL `https://www.ndss-symposium.org/ndss2025/`. To appear.

Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, May 2019. doi: 10.1109/SP.2019.00052.

Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, pages 1051–1065, New York, NY, USA, 2022. Association for Computing Machinery. doi: 10.1145/3548606.3560602.

Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association, 2021. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo`.

Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2021.

Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. ReGuard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 65–68, New York, NY, USA, May 2018. Association for Computing Machinery. ISBN 978-1-4503-5663-3. doi: 10.1145/3183440.3183495. URL `https://dl.acm.org/doi/10.1145/3183440.3183495`.

Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin 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*, pages 531–548, London United Kingdom, November 2019. ACM. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3363230. URL https://dl.acm.org/doi/10.1145/3319535.3363230.

Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, and Chin-Wei Tien. SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 458–465, October 2019. doi: 10.1109/IOTSMS48152.2019.8939256. URL https://ieeexplore.ieee.org/document/8939256/.

Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1795–1809, May 2022a. ISSN 1941-0018. doi: 10.1109/TDSC.2020.3037332. URL https://ieeexplore.ieee.org/document/9256983/.

Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. EthPloit: From Fuzzing to Efficient Exploit Generation against Smart Contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 116–126, London, ON, Canada, February 2020. IEEE. ISBN 978-1-7281-5143-4. doi: 10.1109/SANER48275.2020.9054822. URL https://ieeexplore.ieee.org/document/9054822/.

Imran Ashraf, Xiaoxue Ma, Bo Jiang, and W. K. Chan. GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities. *IEEE Access*, 8:99552–99564, 2020. ISSN 2169-3536. doi: 10.1109/ACCESS.2020.2995183. URL https://ieeexplore.ieee.org/document/9094680/.

Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, Seoul South Korea, June 2020. ACM. ISBN 978-1-4503-7121-6. doi: 10.1145/3377811.3380334. URL https://dl.acm.org/doi/10.1145/3377811.3380334.

Valentin Wüstholz and Maria Christakis. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, Virtual Event USA, November 2020. ACM. ISBN 978-1-4503-7043-1. doi: 10.1145/3368089.3417064. URL https://dl.acm.org/doi/10.1145/3368089.3417064.

Songyan Ji, Jian Dong, Junfu Qiu, Bowen Gu, Ye Wang, and Tongqi Wang. Increasing Fuzz Testing Coverage for Smart Contracts with Dynamic Taint Analysis. In *2021 IEEE 21st*

International Conference on Software Quality, Reliability and Security (QRS), pages 243–247, December 2021. doi: 10.1109/QRS54544.2021.00035. URL https://ieeexplore.ieee.org/document/9724805/. ISSN: 2693-9177.

Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu 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, September 2021. doi: 10.1109/EuroSP51992.2021.00018. URL https://ieeexplore.ieee.org/document/9581164/.

Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239, November 2021. doi: 10.1109/ASE51524.2021.9678888. URL https://ieeexplore.ieee.org/document/9678888/. ISSN: 2643-1572.

Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawende F. Bissyande. SmartGift: Learning to Generate Practical Inputs for Testing Smart Contracts. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 23–34, Luxembourg, September 2021. IEEE. ISBN 978-1-6654-2882-8. doi: 10.1109/ICSME52107.2021.00009. URL https://ieeexplore.ieee.org/document/9609227/.

Wei Zhang. Beak: A Directed Hybrid Fuzzer for Smart Contracts. *International Core Journal of Engineering*, 8(4), April 2022. ISSN 2414-1895. doi: 10.6919/ICJE.202204_8(4).0060.

Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao. xFuzz: Machine Learning Guided Cross-Contract Fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 21(2):515–529, 2024. doi: 10.1109/TDSC.2022.3182373.

Xiaoyin Wang, Jiaze Sun, Chunyang Hu, Panpan Yu, Bin Zhang, and Donghai Hou. EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection. *Wireless Communications and Mobile Computing*, 2022(1):1565007, 2022b. ISSN 1530-8677. doi: 10.1155/2022/1565007. URL https://onlinelibrary.wiley.com/doi/abs/10.1155/2022/1565007. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2022/1565007.

Mitchell Olsthoorn. SynTest-Solidity: Automated Test Case Generation and Fuzzing for Smart Contracts. 2022.

Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, Rochester MI USA, October 2022. ACM.

ISBN 978-1-4503-9475-8. doi: 10.1145/3551349.3560429. URL https://dl.acm.org/doi/10.1145/3551349.3560429.

Songyan Ji, Jin Wu, Junfu Qiu, and Jian Dong. *Effuzz*: Efficient fuzzing by directed search for smart contracts. *Information and Software Technology*, 159:107213, July 2023. ISSN 0950-5849. doi: 10.1016/j.infsof.2023.107213. URL https://www.sciencedirect.com/science/article/pii/S0950584923000678.

Zhenguang Liu, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang. Rethinking Smart Contract Fuzzing: Fuzzing With Invocation Ordering and Important Branch Revisiting, January 2023. URL http://arxiv.org/abs/2301.03943. arXiv:2301.03943 [cs].

Chaofan Shou, Shangyin Tan, and Koushik Sen. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 322–333, New York, NY, USA, July 2023. Association for Computing Machinery. ISBN 979-8-4007-0221-1. doi: 10.1145/3597926.3598059. URL https://dl.acm.org/doi/10.1145/3597926.3598059.

Weimin Chen, Xiapu Luo, Haipeng Cai, and Haoyu Wang. Towards Smart Contract Fuzzing on GPUs. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2255–2272, May 2024. doi: 10.1109/SP54263.2024.00229. URL https://ieeexplore.ieee.org/document/10646637/?arnumber=10646637. ISSN: 2375-1207.

Jiahui Yang, Xiangfu Zhao, Hanfeng Zhang, Long He, Shiji Wang, and Naixiang Gou. CSAFuzzer: Fuzzing smart contracts combining with static analysis. *Empirical Software Engineering*, 30(3):1–29, May 2025. ISSN 1573-7616. doi: 10.1007/s10664-025-10623-3. URL https://link-springer-com.bibliotecauned.idm.oclc.org/article/10.1007/s10664-025-10623-3. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 3 Publisher: Springer US.

Fanglei Shi, Jinsheng Yang, and Zhaohui Guo. TPH-Fuzz: A Two-Phase Hybrid Fuzzing Framework for Smart Contract Vulnerability Detection. *Electronics*, 14(7):1465, April 2025. ISSN 2079-9292. doi: 10.3390/electronics14071465. URL https://www.mdpi.com/2079-9292/14/7/1465.

Ziqiao Kong, Cen Zhang, Maoyi Xie, Ming Hu, Yue Xue, Ye Liu, Haijun Wang, and Yang Liu. Smart Contract Fuzzing Towards Profitable Vulnerabilities, February 2025. URL http://arxiv.org/abs/2501.08834. arXiv:2501.08834 [cs].

Mingxi Ye, Xingwei Lin, Yuhong Nan, Jiajing Wu, and Zibin Zheng. Midas: Mining Profitable Exploits in On-Chain Smart Contracts via Feedback-Driven Fuzzing and Differential Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium*

on Software Testing and Analysis, ISSTA 2024, pages 794–805, New York, NY, USA, September 2024. Association for Computing Machinery. ISBN 979-8-4007-0612-7. doi: 10.1145/3650212.3680321. URL https://dl.acm.org/doi/10.1145/3650212.3680321.

Sourav Chakraborty, N. V. Vinodchandran, and Kuldeep S. Meel. Distinct Elements in Streams: An Algorithm for the (Text) Book, May 2023. URL http://arxiv.org/abs/2301.10191. arXiv:2301.10191 [cs.DS].

Appendix A

Smart contracts: Motivation Examples

This appendix provides the source code of the smart contracts employed in the benchmark suite, which serve as the basis for the experimental evaluation presented in Chapter 4.

```
/* SPDX-License-Identifier: MIT */
pragma solidity ^0.8.0;

// Harvey Fuzzer Motivation Example
// This contract demonstrates the path coverage challenge that Harvey addresses
contract BazExample {
    function baz(int256 a, int256 b, int256 c) public pure returns (int256) {
        int256 d = b + c;

        if (d < 1) {
            if (b < 3) {
                return 1;
            }
            if (a == 42) {
                return 2;
            }
            return 3;
        } else {
            if (c < 42) {
                return 4;
            }
            return 5;
        }
    }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// IFL (Invariant-based Fuzzing Logic) Motivation Example
// This contract demonstrates a crowdsale with an intentional access control vulnerability
contract Crowdsale {
    uint256 goal = 100000 * (10**18);
    uint256 phase = 0;
    // 0: Active, 1: Success, 2: Refund
    uint256 raised;
    uint256 end;
    address owner;
    mapping(address => uint256) investments;

    constructor() {
        end = block.timestamp + 60 days;
        owner = msg.sender;
    }

    function invest() public payable {
        require(phase == 0 && raised < goal);
        investments[msg.sender] += msg.value;
        raised += msg.value;
    }

    function setPhase(uint256 newPhase) public {
        require(
            (newPhase == 1 && raised >= goal) ||
            (newPhase == 2 && raised < goal && block.timestamp > end)
        );
        phase = newPhase;
    }

    function setOwner(address newOwner) public {
        // Fix: require(msg.sender == owner);
        owner = newOwner;
    }

    function withdraw() public {
        require(phase == 1);
        payable(owner).transfer(raised);
    }

    function refund() public {
        require(phase == 2);
        payable(msg.sender).transfer(investments[msg.sender]);
```

```
47          investments[msg.sender] = 0;
48      }
49
50      // Getter functions for better testing
51      function getPhase() public view returns (uint256) {
52          return phase;
53      }
54
55      function getRaised() public view returns (uint256) {
56          return raised;
57      }
58
59      function getGoal() public view returns (uint256) {
60          return goal;
61      }
62
63      function getOwner() public view returns (address) {
64          return owner;
65      }
66
67      function getInvestment(address investor) public view returns (uint256)
        {
68          return investments[investor];
69      }
70
71      function getEnd() public view returns (uint256) {
72          return end;
73      }
74  }
```

Listing A.1: IFL Fuzzer Motivation Example.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3
4  // ItyFuzz Motivation Example
5  // Smart contract with a simple persistent state counter
6  // Demonstrates state-dependent vulnerability detection
7  contract SimpleState {
8      int256 counter = 0;
9
10     // Target value that triggers the bug - let's use 5 as an example
11     int256 constant TARGET = 5;
12
13     // Event to signal when bug is triggered
14     event BugTriggered(int256 counterValue);
15
16     function incr(int256 x) public {
```

```
17          require(x <= counter);
18          counter += 1;
19      }
20
21      function decr(int256 x) public {
22          require(x >= counter);
23          counter -= 1;
24      }
25
26      function buggy() public {
27          if (counter == TARGET) {
28              // In the original paper this was "bug!()" but we'll use
    revert
29              // to make it a detectable vulnerability
30              emit BugTriggered(counter);
31              revert("Bug triggered: counter reached target value!");
32          }
33      }
34
35      // Getter function to check current counter value
36      function getCounter() public view returns (int256) {
37          return counter;
38      }
39
40      // Alternative version that doesn't revert but returns a flag
41      function checkBug() public view returns (bool) {
42          return counter == TARGET;
43      }
44  }
```

Listing A.2: ItyFuzz Fuzzer Motivation Example.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.0;
3
4   // Complex State Contract - Designed to test F0-Estimator vs Voting-based
        pruning
5   // This contract creates many similar but distinct states to favor F0's
        distinctness detection
6   contract ComplexState {
7       // Multiple state variables create combinatorial state explosion
8       uint256 public stateA = 0;
9       uint256 public stateB = 0;
10      uint256 public stateC = 0;
11      uint256 public stateD = 0;
12
13      // Arrays create many similar states with slight variations
14      uint256[10] public arrayState;
```

```
15      mapping(uint256 => uint256) public mapState;
16
17      // Complex state transitions that create redundant paths
18      uint256 public pathCounter = 0;
19      uint256 public redundantCounter = 0;
20
21      // Target conditions for vulnerabilities
22      uint256 constant TARGET_A = 7;
23      uint256 constant TARGET_B = 12;
24      uint256 constant TARGET_SUM = 25;
25
26      event VulnerabilityFound(
27          string vulnType,
28          uint256 stateA,
29          uint256 stateB,
30          uint256 stateC
31      );
32
33      // Function that creates many similar states (voting might not
        distinguish well)
34      function modifyStateA(uint256 increment) public {
35          require(increment > 0 && increment <= 3);
36          stateA += increment;
37          redundantCounter++; // Creates redundant state variations
38
39          // Array modification creates similar but distinct states
40          if (stateA % 2 == 0) {
41              arrayState[stateA % 10] = stateA;
42          }
43
44          checkVulnerabilityA();
45      }
46
47      // Function that creates overlapping state modifications
48      function modifyStateB(uint256 increment) public {
49          require(increment > 0 && increment <= 5);
50          stateB += increment;
51
52          // Create mapping states that are similar but distinct
53          mapState[stateB % 100] = stateB * 2;
54
55          // Cross-dependencies create complex state relationships
56          if (stateB > 5) {
57              stateC += 1;
58          }
59
60          checkVulnerabilityB();
```

```
61      }
62
63      // Function that creates many paths to similar states
64      function redundantPath1(uint256 x) public {
65          require(x <= 10);
66          pathCounter += x;
67          stateC += (x % 3);
68
69          // Multiple ways to reach similar states
70          if (x % 2 == 0) {
71              redundantCounter += 2;
72          } else {
73              redundantCounter += 2; // Same effect, different path
74          }
75      }
76
77      function redundantPath2(uint256 x) public {
78          require(x <= 10);
79          pathCounter += x;
80          stateC += (x % 3);
81
82          // Another way to reach similar states (confuses voting-based)
83          redundantCounter += 2;
84      }
85
86      // Complex state interactions that create subtle distinctions
87      function complexInteraction(uint256 a, uint256 b) public {
88          require(a <= 20 && b <= 20);
89
90          // Create subtle state variations that F0 should detect better
91          stateA += (a % 4);
92          stateB += (b % 7);
93          stateD = (stateA * stateB) % 100;
94
95          // Array and mapping updates create many distinct states
96          arrayState[(a + b) % 10] = stateD;
97          mapState[stateD] = a + b;
98
99          checkComplexVulnerability();
100     }
101
102     // Loop that creates many similar states (tests aggressive pruning)
103     function createSimilarStates(uint256 iterations) public {
104         require(iterations <= 50);
105
106         for (uint256 i = 0; i < iterations; i++) {
107             // Each iteration creates slightly different state
```

```
108             stateC += (i % 3);
109             arrayState[i % 10] = stateC + i;
110 
111             // Redundant calculations create similar but distinct
    execution paths
112             redundantCounter += (i % 2 == 0) ? 1 : 1;
113         }
114     }
115 
116     // Vulnerability checks
117     function checkVulnerabilityA() internal {
118         if (stateA == TARGET_A) {
119             emit VulnerabilityFound("StateA Target", stateA, stateB,
    stateC);
120             revert("Vulnerability A: stateA reached target!");
121         }
122     }
123 
124     function checkVulnerabilityB() internal {
125         if (stateB == TARGET_B) {
126             emit VulnerabilityFound("StateB Target", stateA, stateB,
    stateC);
127             revert("Vulnerability B: stateB reached target!");
128         }
129     }
130 
131     function checkComplexVulnerability() internal {
132         if (stateA + stateB == TARGET_SUM && stateC > 5) {
133             emit VulnerabilityFound("Complex Target", stateA, stateB,
    stateC);
134             revert("Complex Vulnerability: state combination reached
    target!");
135         }
136     }
137 
138     // Getter functions for state inspection
139     function getStates()
140         public
141         view
142         returns (uint256, uint256, uint256, uint256)
143     {
144         return (stateA, stateB, stateC, stateD);
145     }
146 
147     function getArrayState(uint256 index) public view returns (uint256) {
148         require(index < 10);
149         return arrayState[index];
```

```
150     }
151
152     function getMapState(uint256 key) public view returns (uint256) {
153         return mapState[key];
154     }
155
156     function getCounters() public view returns (uint256, uint256) {
157         return (pathCounter, redundantCounter);
158     }
159 }
```

Listing A.3: Modified ItyFuzz Fuzzer Motivation Example.