Static Analysis Approaches for Finding Vulnerabilities in Smart Contracts

by

Asem Abdo Esmail Ghaleb

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

August 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Static Analysis Approaches for Finding Vulnerabilities in Smart Contracts

submitted by **Asem Abdo Esmail Ghaleb** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Electrical and Computer Engineering**.

Examining Committee:

Karthik Pattabiraman, Professor, Electrical and Computer Engineering, UBC
Supervisor

Julia Rubin, Associate Professor, Electrical and Computer Engineering, UBC
Co-Supervisor

Reid Holmes, Professor, Computer Science, UBC
University Examiner

Sidney Fels, Professor, Electrical and Computer Engineering, UBC
University Examiner

Additional Supervisory Committee Members:

Konstantin Beznosov, Professor, Electrical and Computer Engineering, UBC
Supervisory Committee Member

Abstract

The growth in the popularity of smart contracts has been accompanied by a rise in security attacks targeting vulnerabilities in smart contracts, which led to financial losses of millions of dollars and erosion of trust. To enable developers find vulnerabilities in the code of smart contracts, researchers and industry practitioners have proposed several static analysis tools. However, vulnerabilities abound in smart contracts, and the effectiveness of the state-of-the-art analysis tools in detecting vulnerabilities has not been studied.

To understand the effectiveness of the state-of-the-art static analysis tools in detecting vulnerabilities in smart contracts, we propose a systematic approach for evaluating smart contract static analysis tools using security bug injection. We use our proposed approach to evaluate the effectiveness of well-known static analysis tools. The evaluation results show that analysis tools fail to detect significant vulnerabilities and report a high number of false alarms. To improve the state of static analysis for finding vulnerabilities, we expand the space of vulnerability detection and propose static analysis approaches for detecting two-broad categories of vulnerabilities in smart contracts, namely, gas-related vulnerabilities and access control vulnerabilities. Our proposed solutions rely on identifying security properties in the code of smart contracts and then analyzing the dependency of the contract code on user inputs that lead to violating the identified security properties. The results show that our proposed vulnerability detection approaches achieve a significant improvement in the effectiveness of detecting vulnerabilities compared to the prior work.

Lay Summary

Several blockchain systems support running programs, called smart contracts, executed autonomously by the blockchain. Smart contracts are increasingly popular for their ability to streamline financial transactions and automate various processes without intermediaries, such as banks and government institutions. However, they suffer from vulnerabilities (security holes) exploited by malicious attackers for financial gain. The past few years witnessed many attack incidents on smart contracts, resulting in the loss of billions of dollars. This dissertation targets building techniques to discover vulnerabilities in smart contracts to be used by contract developers before running smart contracts on the blockchain. We first study the existing analysis methods to understand their weaknesses and limitations, and we find that they fail to find several vulnerabilities and report high false alarms. To address this, we propose new techniques to discover security vulnerabilities in smart contracts.

Preface

This dissertation is a result of work carried on by the author of the dissertation in collaboration with Dr. Karthik Pattabiraman (supervisor) and Dr. Julia Rubin (co-supervisor). The work presented herein is based on scientific papers published in ISSTA'20, ISSTA'22, and ICSE'23 conferences and a paper under review in an international journal. As the author of the dissertation, in each work, I was responsible for identifying the research ideas, designing the approach, implementing the software tool of the proposed approach, collecting datasets, conducting evaluation experiments, analyzing and presenting results, and writing the paper. Other collaborators were involved in refining the research ideas, editing the manuscripts, and providing feedback and guidance.

The publication details for the work presented in Chapters 4, 5, and 6 are as follows.

- Asem Ghaleb and Karthik Pattabiraman. (2020, July). How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools using Bug Injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020) (pp. 415-427). (Acceptance Rate: 26%).

- Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. (2022, July). eTainter: Detecting Gas-Related Vulnerabilities in Smart Contracts. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022) (pp. 728-739). (Acceptance Rate: 24.5%).

- Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. (2023, May). AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In Pro-

ceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023) (pp. 945-956). (Acceptance Rate: 26%).

- Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. (Under Review). Detecting Gas-Related Vulnerabilities and Code Smells in Smart Contracts. (15 pages).

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my gratitude to my advisor, Karthik Pattabiraman, for giving me the opportunity to join the Dependable Systems lab and for providing unstinting support and invaluable guidance throughout my doctoral research. Karthik, your expertise and commitment to my academic growth have been instrumental in shaping this academic achievement.

I am equally indebted to my co-advisor, Julia Rubin, whose expertise has enriched my research journey significantly. Julia, your profound insights and constructive criticism have been indispensable in refining my work and broadening my perspective as a researcher.

Thank you, Karthik and Julia, for the countless hours spent in discussions, the insightful feedback, and the encouragement that propelled me forward.

I also want to express my deepest gratitude to my mother, brothers, and sisters for their unending love and support, and to my dear wife Bothaina, whose love and encouragement have been a guiding light during difficult moments. Bothaina, I am profoundly grateful for your sacrifices, late-night companionship, and unwavering belief in me.

I would also like to extend my thanks to Rui Xi for his assistance in a couple of evaluation experiments and to all my colleagues in the Dependable Systems lab for their thoughtful discussions and feedback on my work.

Lastly, to all those who have contributed, in any way, to the successful completion of this thesis, your support is greatly appreciated.

Dedication

To my late father Abdo Esmail, my mother Mariam, and my wife Bothaina – Your unwavering love and support illuminated my path throughout this journey.

Chapter 1

Introduction

Blockchain is a type of digital ledger technology that operates in a decentralized manner and enables secure and transparent transactions to occur without intermediaries. The first blockchain system was the Bitcoin cryptocurrency [109], launched in 2009, and since then, the growth of blockchain technology expanded beyond the realm of cryptocurrencies. Blockchain-based applications, nowadays, are being adopted in several industries, such as healthcare, supply chain, and financial services. A major advancement in blockchain technology is the development of blockchain systems supporting smart contracts, e.g., Ethereum [170] and EOS [85].

Smart contracts are programs running on top of a blockchain that can receive and execute transactions autonomously. The growing interest in smart contracts is driven by their inherited decentralized nature that enables secure distributed computations while eliminating the need for trusted third parties [78]. Smart contracts, thus, provide means to automate and host decentralized applications (DApps) [90] and integrate other various innovations with the blockchain, with Ethereum currently being the most popular blockchain platform for running smart contracts.

Ethereum smart contracts are written in high-level Turing-complete programming languages, such as Solidity [41], and they are compiled to Ethereum virtual machine (EVM) low-level bytecode that is deployed to the blockchain. The bytecode deployed on Ethereum permissionless public networks is visible to the public and can be invoked by anyone who has an account on the blockchain [49]. Smart contract invocation requests from users are created as transactions that get added to

the blockchain. Unlike traditional applications, smart contracts store state variables on the blockchain in a native persistent memory area (storage) maintained over executions. Further, the execution of smart contracts on the Ethereum blockchain consumes fees, called gas, paid for by users calling the contract functions [170]. The gas consumed varies for each execution, and depends mainly on the executed bytecode instructions and the state of the contract on the blockchain. A contract execution proceeds as long as the users dedicate enough gas; within the limit set by Ethereum (i.e., block gas limit).

1.1 Problem Setup

Smart contracts are susceptible to bugs and vulnerabilities, just like any other software. However, the impact of exploitable vulnerabilities in a smart contract can be much more significant due to the financial nature of smart contracts, leading to various risks, including financial losses and reputational damage. Ethereum smart contracts, for example, manage accounts holding millions of dollars, and decentralized finance (DeFi) [166, 167] is one of the most common domains where smart contracts are used. This financial nature makes smart contracts an attractive target for attackers seeking to exploit vulnerabilities for financial gain. The past few years have seen several high-profile attacks on smart contracts, resulting in significant financial losses. For example, the DAO vulnerability [40] in 2016 led to the theft of $60 million worth of Ether, while an access control vulnerability in ValueDeFi platform [127] in 2021 resulted in the theft of $10 million. According to [176], about $1.55 billion were stolen from DeFi in 2021 due to vulnerability exploits.

Therefore, it is critical to analyze and test smart contracts thoroughly for vulnerabilities before deployment. The immutability of blockchain transactions makes it difficult to reverse the attack transactions and recover the financial losses. The Ethereum community needs to hard-fork the Ethereum blockchain (i.e., create a new version of the blockchain), as was the case with the DAO attack. However, hard forks for reversing attack losses are controversial as they break the immutability principle of blockchain technology [144]. As a result, the Ethereum community started implementing other actions, such as restricting attacker addresses. However, such restrictions are implemented by certain applications running on top of

the blockchain, e.g., exchanges and lending platforms. Thus, there are no guarantees for recovering financial losses due to vulnerabilities. Subsequently, smart contract attacks due to vulnerabilities can also damage the reputation of smart contracts and the blockchain ecosystem, thereby reducing user adoption and trust.

Test cases can be helpful in detecting vulnerabilities in smart contracts, but they have limitations. The code coverage of test cases is limited due to the difficulty of covering all possible scenarios [80, 91]. Further, several corner cases get overlooked by testing [20], and it is difficult to reproduce all vulnerabilities in a test environment [156]. Additionally, the gas mechanism limits the adoption of runtime verification as instrumenting the code with assertions increases the gas cost [99, 150]. On the other hand, runtime verification approaches that rely on instrumenting EVM clients require hard-forking all the EVM clients [99, 150]. Further, as smart contracts take as input several transactions and maintain a persistent state, it is challenging for dynamic analysis approaches, such as fuzzing, to discover transactions sequences that can alter the persistent state of the contract to trigger vulnerabilities [27, 28]. Approaches based on formal methods, such as model checking, can successfully verify smart contracts but have limitations. For instance, the effectiveness of model checking is restricted by the challenge of precisely modeling the details of smart contract execution on the blockchain environment, such as the gas mechanism and the contract storage, due to the limitations induced by the input language of the model-checkers [111, 150]. Other approaches, e.g., theorem proving, typically involve a combination of automated and human efforts, with human expertise and participation being necessary [150].

Static analysis can enable building fast and fully-automated security analysis techniques for smart contracts that offer a high coverage of smart contract vulnerabilities and high scalability, requiring neither test suites nor human expertise and effort. Static analysis techniques identify potential vulnerabilities in smart contracts by analyzing the code of the smart contract without actually executing it. This can help developers identify and fix vulnerabilities during the development phase, which is especially important given the immutability of smart contracts on the blockchain. Thus, static analysis techniques can provide an additional layer of security for smart contracts, helping to reduce the risk of financial losses due to vulnerabilities or exploits.

The overarching goal of this dissertation is to build static analysis approaches for finding gas-related and access control vulnerabilities in smart contracts. To achieve this, we first understand the state of the current static analysis tools. Based on this understanding, we advance the state of static analysis for smart contracts, and propose new static analysis-based approaches for finding gas-related and access control vulnerabilities in smart contracts. The results show that our proposed approaches are more effective in detecting vulnerabilities than the state-of-the-art static analysis tools.

1.2 Challenges and Motivation

Statically analyzing smart contracts for vulnerabilities is a challenging problem and requires sophisticated techniques to overcome multiple challenges. One of the main challenges is that smart contracts have several peculiarities (Section 2.1) that require consideration by static analysis methods. For example, smart contracts have access to persistent storage on the blockchain, which maintains data over executions [170]. Further, the dependency on the gas concept to execute contracts gives rise to several vulnerabilities due to running out of gas. Detecting these vulnerabilities is complicated as gas varies costs for different instructions based on the contract's current state on the blockchain. How a contract interacts with other contracts on the blockchain further complicates matters.

Furthermore, the Solidity language used for developing smart contracts differs from conventional programming languages targeted by previous static analysis efforts. As an example, Solidity employs an atypical method to implement dynamic data structures, such as arrays, via cryptographic hashing [150]. In this method, offsets of structure's elements within the storage are the hash of the structure offset and element keys, making it difficult to perform essential static analyses such as data-flow analysis. Additionally, Solidity does not have a security model based on permissions. As a result, smart contract developers implement access control checks in an ad-hoc manner based on their judgment. The inconsistency in implementing access control checks creates another obstacle to statically analyzing smart contracts for access control vulnerabilities with the lack of access control specifications.

1.2.1 Understanding the Current State of Static Analysis Tools

Several static analysis tools have been proposed by industry and the research community to statically analyze Ethereum smart contracts for security bugs and vulnerabilities [17, 34, 56, 76, 101, 103, 106, 113, 149, 154].[1] However, there are no studies that have systematically demonstrated the effectiveness of these tools in detecting security bugs [178]. Understanding the state of the current static analysis tools, therefore, will allow us to build more effective static analysis techniques for detecting vulnerabilities in smart contracts than the state of the art.

Research Question 1. *How effective are smart contract static analysis tools in detecting security bugs?*

To address RQ1, we propose *SolidiFI*, a systematic approach for evaluating the effectiveness of smart contracts static analysis tools using security bug injection [64]. The key idea is to introduce security bugs into the contract source code and then use the buggy contracts to study the false-negatives and false-positives of static analysis tools. The results of our evaluation study of smart contract static analysis tools showed that all the evaluated tools have significant false-positives and false-negatives.

Another main finding of our evaluation study of static analysis tools was that existing tools depend mainly on analyzing patterns of the code syntax and symbolic traces without considering root causes in the analyzed code leading to vulnerabilities. We leverage this finding from our evaluation study and target finding vulnerabilities by analyzing smart contracts for security properties where violations of these properties lead to vulnerable code. We focus in this research on gas-related and access control vulnerabilities, which are among the most common and critical vulnerabilities occurring in smart contracts [161]. However, these vulnerabilities have not received much attention from researchers studying smart contracts vulnerability detection.

[1] In what follows, we use the terms security bug and vulnerability interchangeably, as the term security bug is often used in the literature to reference security bugs leading to vulnerabilities.

1.2.2 Detecting Gas-Related Vulnerabilities

When a contract user provides insufficient gas to execute a smart contract or if the gas needed for execution exceeded the block gas limit, the contract execution halts, and the transaction execution gets reverted, i.e., Ethereum rolls back all changes made to the contract state by the transaction, resulting in unwanted behaviors. This can be particularly problematic if functions that exceed the gas limit are responsible for transferring Ether out of the contract. In such cases, contract owners/users may not be able to access their money (Ether) due to the transaction being reverted, e.g., in the Governmental contract [160], about $2.5 million worth of Ether got locked out (cannot withdraw Ether from the contract) due to this issue. Unfortunately, smart contracts may contain code patterns that can cause the contracts to run out of gas, which can be exploited by attackers to induce unwanted behavior, such as Denial-of-Service (DoS) attacks [161]. We call these *gas-related vulnerabilities*.

While there is prior work [76] for statically discovering gas-related vulnerabilities in smart contracts, they typically rely on pre-specified code patterns and rules, which can make them brittle and prone to high false-positives, as even small variations in the code patterns can cause the tools to fail.

Research Question 2. *How can we effectively detect gas-related vulnerabilities in smart contracts?*

In this dissertation, we find that the common root cause of gas-related vulnerabilities is the gas exceptions triggered in the contract code due to the dependency of the contract code on data items either provided or manipulated by the contract users. We use this insight to build *eTainter*, a static taint analysis-based [132] approach to find gas-based vulnerabilities; without any pre-existing code patterns and rules [70]. Our evaluation results show that *eTainter* outperforms prior work with high recall and precision, over 90% each, and *eTainter* has an analysis time of about eight seconds per contract.

1.2.3 Detecting Access Control Vulnerabilities

Access control vulnerabilities are common vulnerabilities in smart contracts [161], where studies show that they form about 23% of real-world exploits [176] in smart contracts. As most smart contracts have a financial nature and handle valuable assets, smart contract developers extensively rely on access control to protect assets managed by smart contracts from being misused by malicious or unauthorized people. As mentioned earlier in Section 1.2, implementing access control checks by developers based on their judgment can be prone to errors and oversights, and it results in several vulnerabilities in smart contracts, called access control vulnerabilities. A known example is a hack that targeted Parity Wallet in 2017 and led to locking out (freezing) about $280M worth of Ether [18]. The inconsistency in implementing access control and the lack of access control specifications for a smart contract makes it difficult to identify access control checks and reasons about whether a contract meets access control needs and is free of access control vulnerabilities. In addition, smart contracts may contain code patterns that look like vulnerabilities but are really part of the contracts' functionality, and it is not straightforward to statically distinguish these patterns from vulnerabilities with the lack of contract access control specifications.

Current approaches for detecting access control vulnerabilities [17, 101] rely on pre-defined rules and patterns, and the contract transactions history to identify access control checks. However, our analysis results show that such pre-defined rules do not generalize well, and the pre-defined patterns are not precise enough, leading to a high number of false alarms and undetected vulnerabilities. Similarly, the dependency on the contract transactions to mine access control is insufficient to mine several access control roles, leaving several vulnerabilities undetected.

Research Question 3. *How can we effectively detect access control vulnerabilities in smart contracts with no contract access control specifications?*

The key observation in this dissertation is that all access control checks have unique functionality that can be identified through data-flow analysis. We use this observation to build the *AChecker* approach [72]. *AChecker* first infers ac-

cess control checks implemented in the contracts using static data-flow analysis, and then analyzes for missing or violated access control checks leading to vulnerabilities. *AChecker* performs further optimizations to distinguish vulnerabilities from contract-intended functionality. We experimentally evaluate *AChecker* and find that *AChecker* outperforms prior work that relies on predefined patterns and transactions history, with 80% recall and 88% precision. In addition, *AChecker* has an analysis time of 11 seconds per smart contract.

1.2.4 Research Workflow

Figure 1.1 shows the workflow of the research conducted in this thesis.

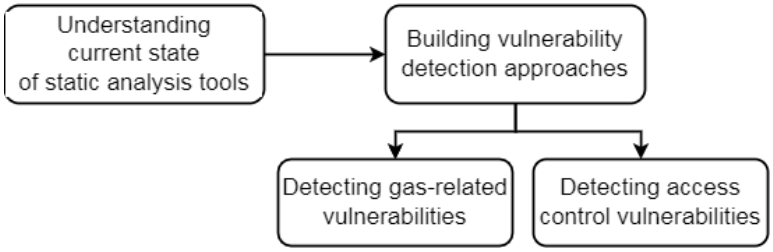

Figure 1.1: Research workflow.

In the beginning, this research aimed to comprehend the existing landscape of static analysis tools for Ethereum smart contracts. This step involved reviewing industry and research-proposed tools for detecting security bugs in smart contracts. Notably, we discovered that while several tools were available, there was a lack of systematic evidence regarding their effectiveness in detecting security bugs. In response to the gap in evidence, we formulated the first research question: How effective are smart contract static analysis tools in detecting security bugs? To address this question, we designed the *SolidiFI* approach, which systematically evaluates the static analysis tools by injecting security bugs into contract source code. This allowed us to systematically study the false-negatives and false-positives of smart contract static analysis tools.

The results of this evaluation study indicated that all the evaluated tools had significant false-positives and false-negatives [64]. Further, we made some observations from the evaluation results. The first key observation was that existing tools

primarily rely on looking for specific predefined vulnerable patterns without thoroughly considering the root causes of vulnerabilities in the code. Second, we found that gas-related and access control vulnerabilities are out of the scope of most of the existing tools, and only a few sub-cases of these vulnerabilities are supported by a few tools, even though these vulnerabilities are common and critical in smart contracts. These findings and observations led us to identify an opportunity for improving the state of static analysis techniques for smart contracts, targeting an avenue gas-related and access control vulnerabilities [63, 70, 72].

Building upon the insights gained from the evaluation study, we formulated the second research question: How can we effectively detect gas-related vulnerabilities in smart contracts? In response to this question, we developed the *eTainter* approach. Unlike prior work that use pre-specified code patterns and rules, *eTainter* utilizes static taint analysis to find gas-based vulnerabilities without any pre-existing code patterns. This approach demonstrated improved effectiveness with high recall and precision, allowing for better detection of vulnerabilities related to gas usage in smart contracts.

Moving on to the third research question: How can we effectively detect access control vulnerabilities in smart contracts with no contract access control specifications? We investigated how to effectively detect access control vulnerabilities in smart contracts without explicit access control specifications. Existing approaches rely on predefined patterns and transaction history, resulting in false alarms and missed vulnerabilities. To address this challenge, we proposed *AChecker* approach. This approach uses static data-flow analysis to infer access control checks in contracts and analyze for missing or violated checks. Further, *AChecker* uses symbolic analysis to differentiate vulnerabilities from intended functionality. *AChecker* demonstrated enhanced effectiveness with significant recall and precision, improving the detection of access control vulnerabilities.

1.3 Contributions

The contributions of this research are as follows.

- Proposes a systematic evaluation methodology of static analysis tools' effectiveness based on security bug injection. We develop an automated tool

of the methodology, *SolidiFI*, that injects security bugs into all possible locations in the source code of smart contracts and uses the generated buggy contracts to evaluate the false-negatives and false-positives of static analysis tools. Using *SolidiFI*, we show that current static tools fail to detect several security bugs, ranging from 129 to 4137 undetected bugs, and report massive false alarms, ranging from 2 to 801. (Chapter 4).

- Formulates the detection of gas-related vulnerabilities as a taint analysis problem. We propose an inter-procedural static taint-analysis approach for smart contract EVM bytecode that considers smart contract new concepts, such as tracking taints through the contract's persistent storage, in addition to using domain-specific optimizations to reduce false-positives. We develop an automated tool of our proposed taint analysis approach, *eTainter*, to detect gas-related vulnerabilities in smart contracts with high precision and recall outperforming prior work. Our study of gas-related vulnerabilities shows that 7% of real-world smart contracts deployed on Ethereum have at least one case of gas-related vulnerabilities. (Chapter 5).

- Proposes a static data-flow analysis-based technique for inferring the access control implemented in smart contracts without relying on either ad-hoc and pre-specified code patterns or existing transactions history. We also propose a symbolic execution-based method that automatically infers cases where untrusted users are allowed to manipulate access control data as an intended functionality of the contract. We build an automated tool, *AChecker*, that combines these proposed techniques to find access control vulnerabilities in smart contracts. *AChecker* flagged vulnerabilities in 624 contracts and 21 most-frequently-used contracts on the Ethereum blockchain. (Chapter 6).

The systematic evaluation methodology of smart contract static analysis tools supports tool developers and researchers to build better detection tools, and also helps smart contract developers choose the most reliable tools with no or low false-negatives and false-positives for their use cases. Furthermore, our proposed vulnerability detection approaches allow developers to find vulnerabilities during the development phase in a fast and effective way before deploying contracts to the

blockchain, which advances the vulnerability detection space and improves the overall security of smart contracts. In addition, proposing vulnerability detection approaches that consider the root causes of vulnerabilities rather than relying on specific pre-defined patterns provides insights for researchers to build better vulnerability detection methods. The three automated tools developed in this research, *SolidiFI* [65], *eTainter* [69], and *AChecker* [67], are freely available as open-source artifacts to be used by Ethereum smart contract developers and security practitioners, as well as to facilitate reproducibility and reusability by future researchers. Finally, several benchmarks collected and generated by this research have also been made publicly available [66, 68, 71] to be used by other studies.

Chapter 2

Background

2.1 Ethereum Smart Contracts

Smart contracts are programs running on top of a blockchain that can receive and execute transactions autonomously without trusted third parties [78]. Ethereum [19], a distributed computing platform running smart contracts, is the most popular blockchain platform for executing smart contracts. Figure 2.1 shows an overview of Ethereum smart contracts generation, deployment, and invocation. Ethereum smart contracts are written in high-level languages, e.g., Solidity [50], Vyper [157], where Solidity is the most used programming language for developing Ethereum smart contracts [21]. For instance, according to DefiLlama [42], a website that tracks the total value locked (TVL) in DeFi protocols, DeFi protocols supported by Solidity smart contracts hold about 91% of the TVL, as of April 2023. Smart contracts get compiled into Ethereum Virtual Machine (EVM) bytecode that is deployed and stored in the blockchain accounts. The EVM bytecode deployed on the blockchain gets executed through transactions upon user requests, and smart contracts can interact with each other. The EVM bytecode is executed by miners,[1] a network of mutually untrusted nodes, and is governed by the consensus protocol of the Ethereum blockchain. Unlike traditional programs, executing the code of a smart contract costs fees, called gas, paid by the user invoking the contract.

[1] In the version of Ethereum employing Proof of Stake (PoS) as a consensus mechanism, the term validators is used instead of miners.

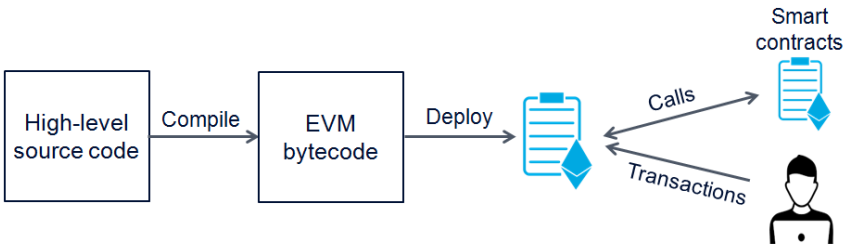

Figure 2.1: Smart contracts overview.

2.1.1 Transactions

In Ethereum, there are two types of accounts, externally-owned accounts (EOA) and contract-based accounts. EOA is an account managed by humans while the contract-based account is associated with a smart contract deployed on the Ethereum network. In Ethereum, users interact with smart contracts through transactions [170]. Transactions are typically encoded and signed calls originating from EOA. For a user to initiate a transaction, the user needs to have an EOA on the Ethereum blockchain where the contract is deployed.

The transaction consists of several fields, such as *from* (sender address), *to* (transaction destination address), *gasLimit*, *maxFeePerGas*, *value*, and *data*. The *gasLimit* field holds the maximum amount of gas (in gas units) the user wants to spend to execute the transaction, while the *maxFeePerGas* indicates the maximum fee (in Ether) the user is willing to pay for each gas unit. The *value* field represents the Ether value transferred between accounts, while the *data* field can be used for other information, such as payment reference. When the transaction is to call a contract function rather than to transfer Ether between accounts, the *value* field can be 0, and the *data* field encodes the function signature (first four bytes) followed by the function arguments. For example, Figure 2.2 shows the format of a transaction for calling the function "*withdraw(uint256 _amount)*" in a contract deployed at the account address "*0xac43eb73b6a9e107830aff4df5077c2b3d481e4a*".

The data associated with the transaction calling the contract (e.g., value, function arguments) is stored in a read-only memory area called `calldata` (*msg.data* in Solidity). This memory area is accessed and read by a set of instructions in

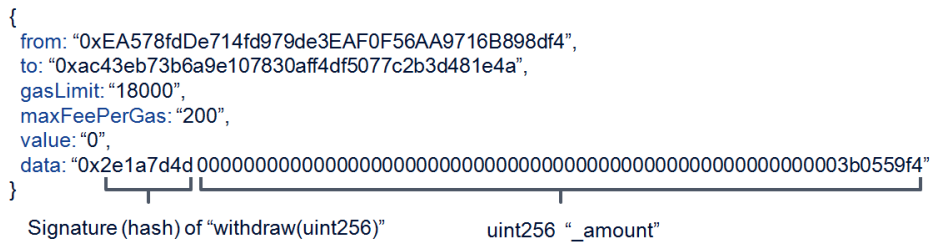

Figure 2.2: Example of a transaction to call a contract function.

the EVM bytecode, as we discuss later in Section 2.1.4. Thus, static taint analyses need to precisely model `calldata`, as data propagating from `calldata` memory area (e.g., function arguments) forms a source of taints.

2.1.2 Gas Mechanism

To prevent abuse of computational resources, Ethereum proposes the gas mechanism to execute smart contracts where miners executing smart contracts receive execution fees called gas for running a transaction, paid upfront (in terms of Ether)[2] by the submitting user [169]. Ethereum assigns a gas cost (in terms of *gas units*) for every low-level bytecode instruction. When a user submits a transaction to call a contract function, they specify the maximum gas they want to spend on executing the invoked contract function and the gas fee (price) for each gas unit (more details in Section 2.1.1). When the execution completes, the user gets charged the execution fee, which is calculated as the total gas cost consumed for the execution multiplied by the gas fee for each gas unit (*execution fee* = *gas cost* $\times$ *gas fee*). The user is refunded the remaining amount if the execution consumed less gas than the maximum gas stated by the user in the transaction.

As shown in Figure 2.3, in the case when the execution runs out of gas (exceeds the gas limit specified by the user), Ethereum aborts the transaction, the execution halts, and the transaction gets reverted (i.e., Ethereum rolls back all changes made before transaction failure). Further, Ethereum imposes an upper bound on the gas that can be consumed by miners to execute transactions in each block, i.e., block gas limit. The block gas limit is a mechanism used to limit the block size and the

[2]Ether is the native cryptocurrency for the Ethereum blockchain.

amount of computations in each block, thereby preventing attacks that consume too many resources, and encouraging developers to write efficient code. Therefore, when the execution gas cost exceeds the block gas limit during the execution of a transaction within a block, the transaction fails, and its execution gets reverted.

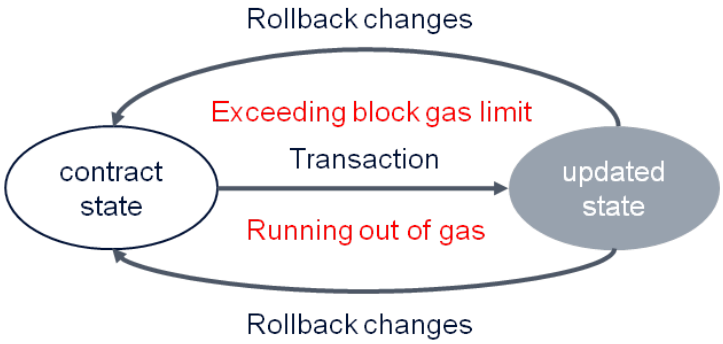

Figure 2.3: Running out of gas behavior.

The gas mechanism introduces an extra layer of complexity to the development of smart contracts. Developers need to carefully write the code to ensure the gas cost of their code is within the limits, and handle exceptions due to running out of gas properly. This could be challenging for developers [178], given that gas is calculated for low-level bytecode instructions, and varies based on the contract state. Further, the abortion of transactions due to running out of gas and exceeding the limits, coupled with the lack of proper handling of these abortions by developers, poses several security risks, such as potential DoS attacks [76, 164, 165].

As the gas cost depends on the contract code and its state (e.g., values of the state variables), attackers can be the contract users. The malicious users may misuse the contract functionality, for instance, to force transactions to specific contract functions to run out of gas and fail indefinitely. To illustrate how the gas mechanism can enable the contract users (wanting to behave maliciously) to abuse the functionality of the contract, Figure 2.4 shows an excerpt of a contract example, in which the function `payUsers` (line 4) is to be used to distribute a specific amount of Ether to all the users of this contract (registered in the array *users*). In this example, any user can be a malicious attacker and prevent the function `payUsers` from completing successfully forever, introducing a DoS attack. This is because

the *transfer* function at line 6 is an external call that transfers Ether to each user account. In Ethereum smart contracts, the *transfer* call will trigger a fallback function in each recipient contract.

```
1  contract distributor{
2     address payable [] private users;
3
4     function payUsers(uint _amount) public onlyAdmin{
5         for(uint i; i < users.length; i++) {
6             users[i].transfer(_amount);
7         }
8     }
9     ...
10 }
```

Figure 2.4: Example of a vulnerability due to gas exceptions.

Therefore, for the malicious user to introduce a DoS attack, they only need to create a small contract and deploy it at their user address registered in the array *users*. As the fallback function of this contract will be executed when the *transfer* sends Ether to this contract, the malicious user can make this *transfer* call fail by increasing the gas cost of the code executed in the fallback function. Figure 2.5 shows an example of a contract with an expensive fallback function. The fallback function (line 4) implements a dummy loop of 10000 iterations, updating the value of a storage variable (*dummy*) in each iteration (writing to storage is an expensive operation, as it updates the blockchain state). This expensive fallback will force the *transfer* call to this contract to run out of gas and fail, and subsequently the `payUsers` function to fail – EVM restricts the gas for each *transfer* call to 2300 gas units. The failure of one transfer call in the function *payUsers* will cause the execution of the whole function to fail, and the changes will be rolled back, resulting in a DoS.

2.1.3 Ethereum Virtual Machine (EVM)

Smart contracts run in a stack-based Ethereum virtual machine (EVM), which maintains a stack of 256-bit words. The smart contract has access to volatile memory that is initialized at the beginning of each execution. Further, in EVM, each

```
1 contract malicious{
2     uint256 dummy;
3
4     function () external payable {
5         for(uint256 i=1; i < 10000; i++) {
6             dummy=i*(i-1);
7         }
8     }
9 }
```

Figure 2.5: Example of a contract with an expensive fallback function.

contract has access to a persistent (non-volatile) private key-value storage of 256-bit keys and 256-bit values, in which data is stored on the blockchain. We refer to each key-value field in the storage as a storage slot. Unlike traditional programs, the contract state variables (variables declared in the global scope of the contract) are stored in the contract storage rather than in the volatile memory, and their values are maintained over multiple executions.

State variables are arranged in the contract storage consecutively based on their order-of-declaration in the contract code. For example, the state variable declared first in the contract gets stored at slot number '0' within the contract storage, and the following state variable occupies slot number '1', etc. The layout of the contract storage for dynamic data structures (e.g., array, mappings) is unconventional when compared with traditional programming languages. For example, since the size of a dynamic array is unknown at compile time, the Solidity programming language uses one slot as the array's identifier within the storage. This slot maintains (stores) the current size of the array. For the array elements, each element occupies a separate slot in the storage with an offset calculated via a Keccak-256 hash in the EVM bytecode. This unconventional method of implementing dynamic data structures presents a challenge for static analyses of EVM bytecode, specifically for data-flow analysis [76, 96]. The hashing method makes it difficult to trace the flow of data through the contract code, and it becomes challenging to map elements of dynamic data structures to their parent structures.

Table 2.1 shows how the storage layout for the contract in Figure 2.6 would look (adapted from [76]). In this example, slot '0' stores the state variable *v1*, slot

'1' is the identifier of the array *arr* and stores the array length, while slot '2' stores the state variable *v2*. Due to the dynamic nature of the array (*arr*), the Solidity compiler uses the Keccak-256 hash function (SHA3 in EVM bytecode) of the array's identifier (slot '1' in our example) to locate the starting offset for the array elements. Matters get complicated when the contract uses multi-dimensional dynamic structures (e.g., nested arrays), where the identifiers of the nested structures are calculated hash values rather than pre-defined constant slot offsets that can be tracked from the code. For example, in the code shown in Figure 2.6, let us assume that the array *arr* is two-dimensional (*address [][] arr*). In this case, slot '1' is the identifier of the array *arr* (maintaining *a.length*), 'SHA3(1)' is the identifier of the first nested array *arr[0]*(maintaining *arr[0].length*), and 'SHA3(SHA3(1))' is the starting offset of the first nested array (storing value of the element *a[0][0]*).

```
1 contract example{
2    uint v1;
3    address [] arr;
4    uint v2;
5    ...
6 }
```

Figure 2.6: An example contract demonstrating the contract storage layout.

Table 2.1: Contract storage layout example.

Solt# (Offset)	Content
0	v1
1	arr.length
2	v2
...	...
SHA3(1)	arr[0]
SHA3(1)+1	arr[1]
SHA3(1)+2	arr[2]

In addition to the challenges posed by the use of unconventional hashing methods to implement dynamic data structures in Solidity smart contracts, using the contract storage in Solidity smart contracts to store state variables on the blockchain poses two more challenges for smart contract code analyses. First, conducting taint

analysis to detect vulnerabilities require capturing this new concept (use of storage) by precisely modeling the contract storage and its persistent nature, and propagating taints through the storage, besides the propagation of taints in the other data locations, e.g., memory and stack.

The second challenge is the modeling of the multi-transactions concept. To exploit vulnerabilities in smart contracts, attackers execute several transactions in a sequence (over multiple executions). Attackers first need to manipulate state variables in the contract storage to activate the vulnerabilities and then call the functions having the vulnerabilities to conduct the attacks. For code analyses to find these types of vulnerabilities, they need to explore a sequence of transactions to reach the vulnerable state. Static taint analyses will need to model propagating taints in the contract storage over multiple executions. Additionally, multi-transactions modeling makes code analysis challenging for symbolic execution-based analyses, as symbolic execution may end up with complex constraints (combining the constraints of all transactions) to be solved and a vast number of paths to explore. Further, dynamic-based analyses need to find the proper sequence of transactions and input values for each transaction to find vulnerabilities.

To explain the multi-transactions concept, in the contract example shown in Figure 2.7, the function *withdraw* is used to withdraw Ether out of the contract, and only authorized users (defined in the mapping *authorizedUsers*) can call it. This contract is vulnerable, where malicious attackers can withdraw Ether from the contract. However, to be able to call the function *withdraw*, the attacker needs to execute a set of transactions in a sequence. First, the attacker needs to make themself an authorized user by calling the function *Authorize*. However, this function can be called only by the admin of the contract. Therefore, second, the attacker needs to make themself an admin by calling the function *changeAdmin*. This function is not protected, so anyone can overwrite the *admin* state variable, storing the address of the contract admin. Finally, the attacker can now call the function *withdraw* and drain the contract Ether. The attacker, therefore, needs to execute this sequence of transactions (*changeAdmin* $\rightarrow$ *Authorize* $\rightarrow$ *withdraw*) to withdraw Ether out of the contract. The code in this example looks simple and tractable by static code analyses, but the problem is much worse in real-world contracts with complex code of the contract functions.

```
contract example_1{
    address admin;
    mapping (address => bool) authorizedUsers;

    function withdraw() {
        require(authorizedUsers[msg.sender]);
        msg.sender.transfer(address(this).balance);
    }
    function Authorize(address _user){
        require(msg.sender == admin);
        authorizedUsers[_user] = true;
    }
    function changeAdmin(address _newAdmin){
        admin = _newAdmin;
    }
    ...
}
```

Figure 2.7: An example contract to illustrate multi-transactions concept.

2.1.4 EVM Bytecode

Unlike traditional programs that have a main entry point, EVM bytecode starts with a function selector, which provides EVM bytecode with multiple entry points. The function selector is like an entry gate for the contract and routes execution to the called external or public function based on matching the static signature of the called function. When calling a contract, if the provided function signature does not match any of the contract's public/external functions, the execution is directed to the fallback function, if any (defined as `function()`). This function is typically used for receiving Ether.

EVM instructions work on data from either the stack, memory, calldata, or storage. As the EVM is stack-based, each instruction in the bytecode that takes arguments, pops its arguments from the stack (except for the PUSH instructions that take immediate arguments). In addition to the stack, EVM uses memory as an input and output buffer for a few instructions, such as `SHA3` that computes the keccak256 hash of data in memory and pushes the result to the stack, where the memory start location and size are read from the stack. Further, EVM uses two instructions (`SLOAD` and `SSTORE`) for managing persistent data in the contract storage. Both instructions get the corresponding storage addresses from the stack, and `SLOAD` pushes the loaded data to the stack.

EVM also has instructions for managing data in memory (`MLOAD` and `MSTORE`), querying blockchain state (e.g., `TIMESTAMP`), writing events to the blockchain (e.g., `LOG1`), and obtaining execution environment information (e.g., the caller of the contract (`CALLER`)). Further, EVM involves a set of arithmetic (e.g., `ADD`, `MUL`) and logic (e.g., `GT`, `EQ`) instructions, control transfer instructions (e.g., conditional jump (`JUMPI`)), other contracts' call instructions (e.g., `CALL`), and instructions that end the execution (e.g., `STOP`) or revert changes made to the contract state during execution (`REVERT`). Table 2.2 summarizes the semantics of common EVM instructions, excluding common instructions for managing the stack (e.g., `PUSHx`, `POPx`, `DUPx`).

The EVM bytecode is low-level with minimal structured language traits, where it has more similarity to machine-specific assembly code than to structured intermediate representations (IR), such as Java bytecode. Due to the characteristics of the EVM bytecode, static analysis at the bytecode level can be challenging [76, 96]. Compared to Java Virtual Machine (JVM) bytecode, the EVM bytecode is not typed, making it difficult for static analyses to reason about the behavior of the code. Additionally, functions in EVM bytecode are fused together, the code does not have the concept of functions (methods), nor does it has function invocation and return instructions to perform intra-contract function calls. Instead, to transfer control, the EVM pushes the return address to the stack and performs a direct jump to the target address of the function in the code. Then to return to the caller function, the EVM jumps back to the block at the return address it pops from the stack. Fusing functions together makes it difficult for static analysis to identify and isolate specific functions within the bytecode. Finally, the EVM bytecode does not have pre-defined constant target addresses for jumps. The bytecode reads jump target addresses from the stack, making it difficult to construct the code control flow. Therefore, with all these characteristics of the EVM bytecode, analyzing the EVM bytecode to find vulnerabilities requires sophisticated approaches to handle these challenges.

Table 2.2: A subset of EVM instructions and their descriptions (Adapted from [54]).

Opcode	Stack Input	Stack Output	Description
STOP			Halts contract execution
ADD	x,y	x+y	Addition operation
MUL	x,y	x*y	Multiplication operation
SUB	x,y	x-y	Subtraction operation
DIV	x,y	x/y	Integer division
MOD	x,y	x%y	Modulo remainder
EXP	x,y	x**y	Exponential operation
LT	x,y	x<y	Less-than comparison
GT	x,y	x>y	Greater-than comparison
EQ	x,y	x==y	Equality comparison
ISZERO	x	x==0	Simple not operator
AND	y,y	x&y	Bitwise AND operation
OR	x,y	x|y	Bitwise OR
XOR	x,y	x ^ y	Bitwise XOR
SHA3	i(offset),len	keccak256(mem[i:i+len])	keccak256 hash
ADDRESS		ADDRESS(this)	Executing contract address
BALANCE	addr	addr.balance	Balance of account
ORIGIN		tx.origin	Execution origination address
CALLER		msg.caller	Caller address
CALLVALUE		msg.value	Value deposited by transaction
CALLDATALOAD	i	msg.data[i:i+32]	Get input data
CALLDATASIZE		msg.data.size	Size of input data
TIMESTAMP		block.timestamp	block's timestamp
GASLIMIT		block.gaslimit	block's gas limit
MLOAD	i(offset)	mem[i:i+32]	Load word from memory
MSTORE	i(offset), value		Save word to memory
MSTORE8	i(offset), value		Save byte to memory
SLOAD	slot	storage[slot]	Load word from storage
SSTORE	slot, value		Save word to storage
JUMP	dest		Unconditional jump
JUMPI	dest, cond		Conditional jump
GAS		GAS()	Remaining gas
LOG0	offset,len		Log record with no topics
LOG1	offset,len, topic0		Log record with one topic
CALL	gas, addr, value, args, argsL,..	call status	Call a function in another contract
DELEGATECALL	gas, addr, value, args, argsL, ..	call status	Call a function in another contract using the caller storage
REVERT	offset, len		Stop execution and revert state changes
SELFDESTRUCT	addr		Destroy the contract and send its balance to addr

2.2 Mutation Testing

Mutation testing [115] is a testing technique with a foundation in software engineering and is used to test the quality of test suites. The goal is to introduce artificial changes or bugs in the software code to generate several mutations of the code and then use these generated code mutations to check if the test cases will detect the introduced mutants. The more mutants detected by the test cases, the higher the effectiveness of the test suites. Although it can be expensive to generate several mutants, mutation testing is an effective testing technique as it helps identify weaknesses in the test suites [115]. Besides evaluating the quality of test suites, mutation testing has also been adapted in some other contexts, such as injecting bugs to evaluate code analyzers [125].

For example, evaluating code analyzers on their effectiveness in finding vulnerabilities is faced with the challenge that benchmarks with vulnerabilities that enable exploring all angles of the problem are scarce [125]. Therefore, bug injection as a testing approach is capable of generating benchmarks for evaluating code analyzers by injecting vulnerable bugs in the software code.

2.3 Taint Analysis

Taint analysis is used in *information-flow based security* to determine if there is a data-flow from low-integrity data (i.e., *sources*) to high-integrity data (i.e., *sinks*) [132]. The sources are typically data that can be manipulated by the user (e.g., data input from users), and the sinks are typically security-sensitive operations (e.g., writing to a database). A flow of data from sources to sinks represents a vulnerability.

There are generally two forms of tainting: (1) explicit, and (2) implicit. Explicit tainting considers only direct data-flow from sources to sinks, without considering control-flow. Implicit tainting also considers indirect tainting via control-flow and is needed for sound analysis. However, tracking implicit taints is typically more expensive and hence most tainting techniques perform explicit tainting.

Finally, taint analysis can be performed either statically or dynamically. Static taint analysis techniques may be imprecise (i.e., has false-positives), but are typically sound (i.e., cover all potential taint flows in the program and hence have no

false-negatives). Dynamic taint analysis typically is precise (i.e., has no false positives), but its coverage is limited by the inputs provided for executing the program, making it unsound (i.e., has false negatives).

2.4 Formal Methods

In the field of software engineering, formal methods refer to the set of techniques that involve the use of mathematical models, formal languages, and logical reasoning to ensure that software systems meet their intended requirements, are free of errors, and are secure [87, 117]. To verify the security of software systems, formal methods are used to specify security requirements, then developers can ensure that the design and implementation of the system meet these requirements.

Several formal methods are commonly used for software security verification, including model checking [29] and theorem proving [117]. Model checking involves thoroughly examining all potential states of a software system to determine if they meet a set of predetermined properties. A formal model of the software that describes its behavior is constructed as a finite-state automaton, and the specifications of the properties are expressed in temporal logic [117]. Subsequently, a model checking algorithm explores all feasible paths through the model, beginning at an initial state, while confirming if each path adheres to the specified properties. The model checker generates counterexamples for the properties that do not hold.

Theorem proving is typically the use of mathematical logic to verify whether a software system meets specific properties. In practice, the developer begins by defining the intended properties of the software system. These properties can be expressed using a mathematical language such as predicate calculus or temporal logic. Subsequently, the developer creates a formal model of the software system that accurately captures its behavior. Using this formal model, the developer constructs a formal proof to demonstrate that the software system meets the specified properties. The construction of this proof can be carried out using either manual reasoning or automated tools, depending on the complexity of the system and the properties under consideration. Finally, the correctness of the proof can be verified using either a proof assistant or a theorem prover.

Chapter 3

Related Work

In this chapter, we first discuss prior studies on the evaluation of bug-finding tools. Then we discuss related work on smart contracts analysis and verification. Finally, we talk about other efforts targeting security of smart contracts and the Ethereum blockchain ecosystem.

3.1 Bug-Finding Tools Evaluation

The approach of injecting bugs for evaluating the effectiveness of bug finders has been applied in other contexts than smart contracts. Bonett et al. proposed μSE [15], a mutation-based framework for the evaluation of Android static analysis tools that works as follows. First, a fixed set of security operators are created describing the unwanted behavior that the tools being evaluated aim to detect (e.g., data leakage). Then, μSE inserts the security operators into mobile apps based on mutation schemes that consider Android abstractions, tools reachability, and security goals of the tools, thereby creating multiple mutants that represent unwanted behavior within the apps. The mutated apps are analyzed using the static tools to be evaluated. However, unlike our work, the undetected mutants are analyzed manually. Further, their framework focuses only on data leak detection tools, unlike our work, which is more general.

Pewny et al. [125] find potential vulnerable locations in C code and modify the source code to make it vulnerable. Program analysis techniques are used to

find sinks in the programs matching specific bug patterns and find connections to user-controlled sources through data-flow. The program is modified accordingly to make it exploitable. In contrast to our approach, the vulnerable code locations to be injected are randomly chosen, and the implemented prototype targets the injection of spatial memory errors through the modification of security checks.

Dolan-Gavitt et al. [45] propose LAVA for generating and injecting bugs into the source code of programs using dynamic taint analysis. A guard is inserted for every injected bug for triggering the vulnerability if a specific value occurs in the input. Specifically, LAVA identifies an execution trace location where an unmodified and dead input data byte (DUA) is available. The code is added at this location to make the DUA byte available and use it to execute the vulnerability. Unlike our work, the injection is based on dynamic taint analysis, and the injected bugs are accompanied by triggering inputs. In contrast, our goal is to transform invulnerable code to systematically introduce vulnerabilities in it.

In the context of smart contracts, Akca et al. [2] propose a tool to compare the effectiveness of their introduced smart contracts static analyzer with some other tools by injecting a single bug into the contract code. Unlike our approach that injects exploitable security bugs into all potential locations in the contract code, the tool uses Fault Seeder [122] to generate contract mutants by injecting only a single bug snippet (hard-coded in the source code) into a specific location in the smart contract. In addition, the authors conducted a manual inspection of the tool reports to determine false negatives. Injecting a single bug does not provide a comprehensive coverage evaluation of static analysis tools. Also, it is not clear how to evaluate the efficacy of static analysis tools on detecting deep vulnerabilities and corner cases by injecting only a single bug. As presented before, each bug can be introduced in the code in several ways, in this case, injecting a single bug will not test the efficacy of the static analysis tools to detect various variants of each bug.

Concurrent with our work, Durieux et al. [46] compare a set of nine smart contract static analysis tools. Unlike our work, the evaluation is based on using 69 manually annotated smart contracts with 112 security bugs in total. The vulnerable contracts are collected from online repositories that are not agnostic of the evaluated tools; hence, results might be biased (e.g., collecting 50% of the contracts from the SWC Registry referenced by Mythril and maintained by the team behind

it). In contrast, our goal is to perform systematic and comprehensive coverage evaluation of static analysis tools by transforming invulnerable code to systematically introduce vulnerabilities into all valid locations. We evaluated six tools on detecting about 9369 distinct security bugs. To provide a fair evaluation of the tools, we evaluate each tool only on the security bugs that it is designed to detect. Our proposed approach can be easily used to evaluate smart contract static analysis tools for detecting other security bug types. Further, it enables end-users to choose any dataset of smart contracts for evaluating the tools.

There have been few other papers on testing smart contracts. Wu et al. [171] produce test cases by mutating specific patterns in smart contracts. Wang et al. [159] target the generation of test suites for smart contracts. This work guides automatic generation of test cases for Ethereum smart contracts. Eth-mutants [14] is another mutation testing tool for smart contracts. However, it is limited to replacing the comparison operators $<$ to $\leq$, and $>$ to $\geq$ (and vice versa). Moreover, unlike our focus on evaluating smart contracts' static analysis tools, these papers target the generation of test cases for smart contracts.

Following our work, several papers have studied smart contract analysis tools [43, 55, 97, 129]. Bruno et al. [43] evaluate the effectiveness of three verification tools. Kushwaha et al. [97] presented a systematic review of static and dynamic analysis tools for smart contracts. Faura et al. [55] investigate six static analysis tools on their usefulness from a DeFi user perspective. Unlike our work that targets systematic and automated evaluation of contracts' static analysis tools, the goal of these studies is to either review smart contract analysis tools or evaluate them using custom datasets. Further, Ren et al. [129] evaluate nine static analysis tools. This work transcends our work by presenting a set of standards that should be considered for a fair evaluation of static analysis tools, and it also evaluates the tools using different datasets (one of them is the dataset generated in our evaluation study). Finally, Vidal et al. [155] propose an approach to inject certain faults into smart contracts to study the impact of faults on the reliability of smart contracts.

3.2 Smart Contract Vulnerability Detection

3.2.1 Detection of Gas-Related Vulnerabilities

There has been limited prior work on finding gas-related vulnerabilities in smart contracts. One of the first tools to identify gas-related vulnerabilities, MadMax [76], statically analyzes smart contracts for three types of vulnerabilities: (1) unbounded loops, (2) DoS with Failed Call (wallet griefing), and (3) Induction Variable overflows. MadMax constructs an intermediate representation (IR) from the EVM bytecode of the contract, and uses Datalog-based inference rules for extracting contract properties. However, MadMax has three drawbacks. First, it mainly searches for specific vulnerability patterns rather than reasoning on the causes of the vulnerabilities being detected. Thus, even a slight variation in the code syntax of the gas-related vulnerabilities would not be covered by these rules (Section 5.2). Second, MadMax coarsely over-approximates many of the vulnerability cases, e.g., considering a loop that references the size of any storage array in its condition as an unbounded loop. This over-approximation results in high false positives (Section 5.6.2). Finally, MadMax only partially models the contract's storage and memory; and hence, overlooks vulnerabilities that depend on memory data items and storage complex structures, leading to false-negatives.

Following our work, Gas Gauge [110] targets a single class of gas-related vulnerabilities (unbounded loops). Gas Gauge statically uses the control flow graph (CFG) to find loops implemented in the contract source code and the variables affecting the loop conditions. Then it utilizes fuzzing to generate input values to test functions with loops to trigger out-of-gas exceptions. However, unlike *eTainter*, focusing on unbounded loops that can lead to DoS attacks, Gas Gauge considers any loop that depends on data inputs of a function as a vulnerability. This can result in high false-positives since these loops are not always vulnerabilities, as direct inputs are volatile and changeable over executions; hence they do not lead to a DoS. Further, Gas Gauge only performs fuzzing for individual functions having loops. However, loops causing DoS may get bounded by storage data items manipulated by contract users through other functions. Thus, triggering such vulnerabilities requires fuzzing and executing a sequence of functions, which is difficult to achieve.

GasReducer [25] is a static analysis tool that performs bytecode optimization by finding unoptimized sequences of EVM instructions and replacing them with optimized sequences. GASPER [23], and its extension GasChecker [26], focus on optimizing gas consumption in smart contracts by detecting gas-inefficient code patterns. However, these tools do not consider gas-related vulnerabilities and are limited to dead code elimination and simple loop optimizations.

GasFuzzer [5] uses fuzzing to generate inputs for smart contracts and create transactions leading to high gas consumption values. It then analyzes the execution logs for exception disorder vulnerabilities (unhandled exceptions, including those caused by insufficient gas). However, GasFuzzer fuzzes individual functions in smart contracts through single transactions, and many gas-related vulnerabilities only arise when calling multiple functions.

Taint analysis has been used by some tools for finding security bugs in smart contracts. However, none of the taint analysis tools consider gas-related vulnerabilities. Osiris [151] uses static taint analysis to validate arithmetic bugs detected through symbolic execution. Its taint analysis is built on top of symbolic execution and performed on the instructions of the counterexamples generated by the symbolic execution, which makes its approach insufficient for detecting gas-related vulnerabilities as the taint propagation will be limited to the generated counter examples. Ethainter [17] uses Datalog rules to perform information flow analysis to detect composite vulnerabilities. Unlike *eTainter*, which works on tracking taints in EVM bytecode, Ethainter designed an abstract input language to capture information flow semantics in smart contracts, customized to detect composite vulnerabilities, as we will discuss later in this chapter. Finally, Sereum [130] uses dynamic taint tracking to protect against Reentrancy attacks by extending the Ethereum client to perform runtime monitoring. However, like all dynamic analysis techniques, this approach has the drawback of finding only the vulnerabilities that are exercised by the provided inputs.

3.2.2 Detection of Access Control Vulnerabilities

Over the past few years, several tools targeted detecting access control vulnerabilities [17, 34, 96, 101, 106, 113, 149, 154]. The most relevant prior work on detecting

access control vulnerabilities are Ethainter [17] and SPCon [101]. Ethainter was the first tool to focus on detecting composite vulnerabilities (i.e., access control vulnerabilities). Ethainter statically analyzes smart contracts for composite vulnerabilities by performing information-flow analysis using inference rules of a designed abstract input language of the bytecode. However, Ethainter's approach has several drawbacks. First, Ethainter's inference rules expect contracts to be coded in a specific style; hence the access control checks detected by these rules are rather rigid [136], which results in several false-negatives and false-positives. Second, Ethainter's rules coarsely over-approximate defining access control checks, thereby defining several irrelevant conditions as access control checks, such as arithmetic checks on the contract caller's data stored in mapping data structures, resulting in a high false-positive rate.

Unlike Ethainter, SPCon does not rely on specific code patterns but rather analyzes smart contracts for permission bugs through mining access control roles using the contract transaction history available on the blockchain. To mine access control roles, SPCon assumes that the analyzed contract have transaction record for the various contract functions; however, not all contract functions get called and have transactions. This is because these functions only get called in specific circumstances, e.g., to remove smart contracts from the blockchain. Further, SPCon assumes each transaction is benign and is performed by an authorized account of the function, as per the desired contract security policy. However, this cannot be guaranteed, especially for smart contracts having access control vulnerabilities. Thus, the dependency on the transactions history to mine access control roles makes SPCon fails to analyze a large share of smart contracts and unable to mine several access control roles; hence it leaves several bugs undetected (as we will show later in Chapter 6).

Several other tools use static analysis to detect general security bugs in smart contracts, including cases of access control vulnerabilities. Securify [154] checks for the presence or lack of specific code patterns of access control before some code statements, such as those that kill the contracts, through checking for compliance or violation of pre-defined security patterns. Maian [113] only analyzes for suicidal and prodigal contracts in which attackers can kill or steal Ether. Smartcheck [149] is an AST-based approach that employs XPath pattern matching to detect several

bugs. However, only a few cases of access control vulnerabilities can be expressed as XPath patterns. Slither [56] performs taint-analysis in the contract source code to detect several bugs. However, Slither highly over-approximates analysis; hence, it reports a high number of false positives.

Other tools use symbolic execution to find vulnerabilities. Mythril [34] uses symbolic execution to detect a large group of security bugs, including cases of access control bugs. However, the need to execute multiple functions in sequence to reach the vulnerable state and detect vulnerabilities results in false-negatives. Manticore [113] is another symbolic analysis tool for smart contracts. However, similar to Mythril, its detection capability is limited by the number of transactions needed to trigger bugs, and it only looks for arbitrary code execution through `delegatecall`. Finally, teEther [96] employs symbolic execution to generate exploits for a group of access control-similar vulnerabilities; however, combining symbolic paths of several functions to generate exploits makes teEther scale to only a small fraction of Ethereum smart contracts [17].

3.2.3 Other Analysis and Verification Approaches for Smart Contracts

There has been other work on analyzing and verifying smart contracts for other classes of security bugs than gas-related and access control vulnerabilities. We will classify them below based on their underlying techniques.

Symbolic Execution. In addition to Mythril [108], Manticore [105], Maian [113], teEther [96], and Osiris [151] aforementioned, Oyente [103] is the first tool that employs symbolic execution to detect four vulnerability classes; namely, reentrancy, transaction order dependency, mishandled exceptions, and timestamp dependencies. EthIR [3], an extension of Oyente, performs a high-level analysis of EVM bytecode and simplifies the analysis by generating an intermediate rule-based representation where local variables are introduced to each basic block. VerX [124] combines symbolic execution with abstract interpretation. Symvalic [136] employs a static analysis approach that combines value-flow fixpoint computation and symbolic reasoning. SmartScopy [58] analyzer employs symbolic execution to find predefined vulnerable patterns as well as user-defined patterns. Symbolic execution-based approaches usually suffer from the path explosion problem, which

makes them unable to explore all transaction sequences [64, 150]. Unfortunately, the detection of several vulnerabilities in smart contracts, such as gas-related and access control vulnerabilities, often requires a complex sequence of transactions to reach the vulnerable state.

Dynamic Analysis Techniques. Dynamic analysis has been used widely in other contexts. However, more effort is required to set up the execution environment in the context of Ethereum smart contracts, as the blockchain immutability and the need for gas fees discourage the use of the Ethereum networks for testing [108]. Multiple dynamic analysis approaches focused on runtime monitoring than the verification of smart contracts [7, 37, 79]. Another study proposes Dynamit [48], a dynamic analyzer that uses machine learning to classify transaction data and detect reentrancy vulnerabilities. Fuzzing-based methods have been used extensively in the scope of dynamic analysis of smart contracts. These approaches focus on generating inputs to trigger code vulnerabilities [27, 38, 57, 89, 112, 173]. Contractfuzzer[89], the first work that employs fuzzing for smart contract, is a black-box fuzzer that uses the Application Binary Interface (ABI) specifications of vulnerable smart contracts to generate exploits (fuzzing inputs) for two vulnerabilities. SMARTSCOPY [57] also synthesizes adversarial contracts for exploiting vulnerabilities in contracts based on ABI specifications of the contracts covering a larger set of vulnerabilities than Contractfuzzer. Echidna [38] supports grammar-based fuzzing to generate transactions to test smart contracts. The sFuzz [112] adopts the idea of branch distance feedback for exploring hard-to-reach branches. ReGuard [100] targets finding potential reentrancy vulnerabilities through generating possible attacks of random transactions. Christakis [172] targets extending greybox fuzzing by learning new inputs from previous smart contract executions.

Furthermore, SMARTIAN [27] uses guided fuzzing to detect a set of smart contract bugs, where it performs dynamic dataflow analysis to guide the fuzzing engine and to implement bug oracles. The use of SMARTIAN for static taint analysis is limited to initial seed construction to find the set of functions in the contract that include sender-checker routines. Bran [173] is another work that employs static analysis to guide grey-box fuzzing. ContraMaster [158] also guides the generation of transaction sequences for fuzzing through the use of control-flow, data-flow, and the state of the contract. Finally, Effuzz [88] focuses on addressing hard-to-cover

branch constrains. In smart contracts, dynamically detecting vulnerabilities that require executing a massive number of transactions to reach the vulnerable state and trigger the vulnerability, e.g., unbounded loops, could be challenging.

Code-Similarity Based Methods. Other studies employ similar-code matching techniques to detect vulnerabilities in smart contracts. Vulpedia [174] extracts vulnerability signatures from vulnerable and benign contracts, and it then constructs vulnerability detection rules from the extracted vulnerability signatures to build a detection method for smart contract vulnerabilities. SMARTEMBED [62] encodes bug patterns into numerical vectors using structural code embedding techniques, and identifies potential bugs through similarity checking among the numerical vectors.

Formal Methods. Multiple studies target verifying smart contracts against written formal specifications. Nehai et al. [111] are the first to propose a modeling method to verify the implantation of a smart contract against its specification using NuSMV. Kongmanee et al. [94] use model checking-based method to check implementation issues leading to specific breaches for a use case contract. Frank et al. [60] proposes a symbolic execution-based bounded model checker, ETHBMC, for smart contracts. ETHBMC generates exploits that target inter-contract vulnerabilities, specifically for stealing Ether and destroying contracts. Authors in [4, 11, 78, 82, 83] propose semi-automated approaches to analyze smart contracts using interactive theorem provers, such as Isabelle/HOL [86], F* [61], Why3 [168], and K [59]. Some other studies target the challenge of fully formal reasoning of Ethereum smart contracts. Hirai [83] define a formal model for EVM using the Lem language to prove the safety properties of smart contracts that can be checked using interactive theorem provers such as Coq and Isabelle/HOL. Building on this work, Amani et al. [4] extend the existing EVM formalization covering correctness properties. Hildenbrandt et al. [82] and Grishchenko et al. [78] target defining formal semantics of the EVM using K framework [131] and F* functional language, respectively. These semantics enable formal verification of arbitrary properties; however, nontrivial to fully automate. Bhargavan et al. [11] present a work that partially targeted translating Solidity and EVM code to F* language to verify for safety and functional correctness. The authors suggest that such reasoning may require manual proofs. Park et al. [119] propose a verification tool for

the EVM bytecode adopting the EVM semantics presented in [82]. Unfortunately, approaches employing formal methods for smart contracts are limited to certain type of properties, and their effectiveness is restricted due to the challenges of modeling smart contract peculiarities (e.g., gas and the persistent state) [111, 150]. Further, most of the proposed approaches are semi-automated and require manual efforts [150].

3.2.4 Blockchain Security Defenses

Besides studies focusing on smart contract analysis and verification, other efforts have been undertaken to enhance the security of the Ethereum ecosystem, including the proposal of new programming languages and the implementation of various countermeasures [22, 150].

Several high-level languages have been proposed to address Solidity issues. Vyper language [157] adds new features (e.g., bounds checking) and removes some of Solidity's functionalities (e.g., infinite loops) to target vulnerabilities like DoS due to unbounded loops. Flint language [134] introduces an "Asset" type and caller restrictions to protect smart contracts against unauthorized access and to get rid of other vulnerability classes, such as reentrancy and integer overflow/underflow. Bamboo language [13] utilizes polymorphism to mitigate the risk of transaction-ordering dependence vulnerability, along with eliminating reentrancy and DoS due to unbounded loops vulnerabilities.

Further, various countermeasures have been proposed to enhance the security of smart contracts and the Ethereum blockchain. Chen et al. [24] propose an adaptive gas cost mechanism to defend against DoS attacks due to under-priced opcodes. Kosba et al. [95] introduce a cryptographic mechanism to hide transaction data and protect against transaction-ordering dependence and confidentiality failure vulnerabilities. Zhang et al. [175] proposed Town Crier, an authenticated data feed system, to mitigate the randomness generation vulnerability in smart contracts accessing external data. Adler et al. [1] extend Town Crier with a voting-based decentralized oracle to address the centralized point-of-failure present in Town Crier. These countermeasures collectively aim to strengthen the security of smart contracts and the Ethereum network.

3.3 Summary

There is a lack of a systematic approach for evaluating the effectiveness of smart contracts analysis tools in detecting security bugs. Our goal is to bridge this gap and propose a systematic approach for evaluating smart contract static analysis tools using security bug injection.

For vulnerability detection in smart contracts, there has been a vast body of work on statically analyzing smart contracts for classes of security bugs [3, 17, 58, 60, 96, 103, 106, 113, 124, 136, 149, 151, 154, 177], and other approaches for testing smart contracts for security bugs [27, 38, 57, 89, 112, 173]. However, few studies [76, 110] targeted finding gas-related vulnerabilities in smart contracts. Other gas-related studies [23, 25, 25, 26] focused on optimizing the contract gas cost rather than considering gas-related vulnerabilities. Similarly, two studies [17, 101] focused on finding access control vulnerabilities, and other work [34, 96, 106, 113, 149, 154] focused on finding general security bugs including few instances of access control vulnerabilities. The common drawback of these static analysis tools is that these tools rely on ad-hoc predefined patterns to detect vulnerabilities, resulting in many false-negatives and false-positives. Additionally, relying on contract transactions history to find access control vulnerabilities fails to detect many vulnerability cases and results in high false-positives. We aim to fill this gap by proposing effective approaches for finding gas-related and access control vulnerabilities, targeting the root causes of vulnerabilities without relying on predefined patterns or contract transactions history.

In the following chapter (Chapter 4), we present our proposed approach, *SolidiFI*, to systematically evaluate the effectiveness of static analysis tools on their detection of security bugs.

Chapter 4

Evaluating Smart Contract Static Analysis Tools using Security Bug Injection

In this chapter, we study the effectiveness of smart contracts static analysis tools. We first define the problem and highlight the challenges of systematically evaluating the effectiveness of existing smart contract static analysis tools. Then we propose a systematic evaluation approach, *SolidiFI*, that evaluates static analysis tools by injecting security bugs into the smart contract source code. We discuss a set of common security bugs occurring in smart contracts and demonstrate the three different approaches employed by *SolidiFI* to inject security bugs into all potential locations in the code. After that, we design experiments to evaluate the effectiveness of six known static analysis tools on their detection of bugs that they are supposed to detect using *SolidiFI*. We also discuss the drawbacks and limitations of the evaluated tools based on the evaluation results. Finally, we discuss the implications of the proposed approach and the evaluation study.

4.1 Introduction

There have been many security bugs in smart contracts that have been maliciously exploited in the recent past [12, 40, 104]. On the other hand, several approaches and

tools have been developed that statically find security bugs in smart contracts [56, 103, 106, 108, 149, 154]. Unfortunately, many of the static analysis tools have been evaluated either only by their developers on custom datasets and inputs, often in an ad-hoc manner, or on datasets of contracts with a limited number of bugs (112 bugs [47] and 10 bugs [118]). *To the best of our knowledge, there is no systematic method to evaluate static analysis tools for smart contracts regarding their effectiveness in finding security bugs.*

Typically, static analysis tools can have both false-positives and false-negatives. While false positives are important, false negatives in smart contracts can lead to critical consequences, as exploiting security bugs in contracts usually leads to loss of Ether (money). Also, empirical studies of software defects in the field have found that many of the defects can be detected by static analysis tools in theory, but are not detected due to limitations of the tools [147]. In our work, we focus mostly on the undetected security bugs (i.e., false negatives), though we also study false-positives of the analysis tools.

We perform security bug injection to evaluate the false-negatives of smart contract static analysis tools. This problem is challenging for two reasons. First, smart contracts on Ethereum are written using the Solidity language, which differs from conventional programming languages typically targeted by mutation testing tools [41]. Second, because our goal is to inject security bugs, the bugs injected should lead to exploitable vulnerabilities.

This chapter proposes *SolidiFI*,[1] a methodology for systematic evaluation of smart contracts' static analysis tools to discover potential flaws in the tools that lead to undetected security bugs. *SolidiFI* injects security bugs formulated as code snippets into *all* possible locations into a smart contract's source code written in Solidity. The code snippets are vulnerable to specific vulnerabilities that can be exploited by an attacker. The resulting buggy smart contracts are then analyzed using the static analysis tools being evaluated, and the results are inspected for those injected security bugs that are not detected by each tool - these are the false-negatives of the tool. Because our methodology is agnostic of the tool being evaluated, it can be applied to any static analysis tool that works on Solidity.

[1] *SolidiFI* stands for Solidity Fault Injector, pronounced as Solidify.

We make the following contributions in this chapter.

1. Design a systematic approach for evaluating false-negatives and false-positives of smart contracts' static analysis tools.
2. Implement our approach as an automated tool, *SolidiFI*, to inject security bugs into smart contracts written in Solidity.
3. Use *SolidiFI* to evaluate six static analysis tools of Ethereum smart contracts for false-negatives and false-positives.
4. Provide an analysis of the undetected security bugs and false-positives for the six tools, and the reasons behind them.

The results of using *SolidiFI* on 50 contracts show that all of the evaluated tools had significant false-negatives ranging from 129 to 4137 undetected security bugs across seven different bug types despite their claims of being able to detect such security bugs, as well as many false positives ranging from 2 to 801. Further, many of the undetected security bugs were found to be exploitable when the contract is executed on the blockchain. Finally, we find that *SolidiFI* takes less than one minute to inject security bugs into a smart contract (on average). Our results can be used by tool developers to enhance the evaluated tools and by researchers proposing new bug-finding tools for smart contracts.

4.2 Background

4.2.1 Solidity Smart Contract Example

We illustrate smart contracts through a running example, written in Solidity, shown in Figure 4.1. We also use this running example later to show example cases of vulnerabilities.

This contract implements a public game that enables users to play a game and submit their guesses or solutions for the game along with some amount of Ether. The Ether submitted by each user gets added to the contract balance, and if the guess submitted by a user is correct, this user gets assigned as the winner of the game, where the winner can claim all the money in the contract balance.

```
1  pragma solidity =0.5.0;
2
3  contract EGame{
4     address payable private winner;
5     uint startTime;
6     bytes32 solution;
7     bool public claimed = false;
8
9     constructor(bytes32 _solution) public{
10       winner = msg.sender;
11       startTime = block.timestamp;
12       solution = _solution;
13    }
14
15    function play(bytes32 guess) payable public {
16       require (!claimed);
17       require(msg.value >= 500);
18       if(keccak256(abi.encode(guess)) == solution){
19          if (startTime + (5 * 1 days) == block.timestamp){
20              winner = msg.sender;
21           }
22        }
23    }
24
25    function getReward() public {
26       require (!claimed);
27       winner.transfer(address(this).balance);
28       claimed = true;
29    }
30 }
```

Figure 4.1: Simple contract written in Solidity.

The contract declares four state variables (lines 4-7), `winner` of type address, `startTime` of type unsigned integer, `solution` of type bytes, and `claimed` of type Boolean. These state variables are stored on the blockchain in the contract's persistent storage, so the values of these variables are maintained over executions. The constructor at line 9 runs only once when the contract is created, and it sets the initial winner (line 10) to the owner of the contract defined by the account of the user who submitted the create transaction of the contract (`msg.sender`). It also initializes the `startTime` state variable (line 11) to the current timestamp during the contract creation, and the `solution` state variable (line 12) to the hash value of the game solution received as a parameter (`_solution`).

The function `play` at line 15 is called by the user who wants to submit their guess. This function is a `payable` function; hence, users have to transfer Ether when calling it, and the received Ether gets added to the contract balance. The `require` statement at line 16 checks that the current winner has not already claimed the game reward and the game is still active; otherwise the execution of `play` fails and gets reverted. The `require` statement at line 17 also checks that the Ether amount sent when calling `play` (`msg.value`) is not less than 500 Wei.[2] The code at line 18 then compares the hash of the received guess with the solution value. If the comparison is successful, it sets the `winner` to the address of the user's account who called this function (`msg.sender`), provided the guess was submitted within five days of creating the contract. Finally, the function `getReward` at line 25 first checks that the game reward had not been claimed before (line 26). It then transfers the amount of the contract balance (`this.balance`) to the current winner (line 27) and closes the game by setting the `claimed`) state variable to true (line 28).

4.2.2 Static Analysis Tools

We consider six static analysis tools for finding security bugs in smart contracts in this study, Oyente [103], Securify [154], Mythril [108], Smartcheck [149], Manticore [106], and Slither [56]. They all operate on smart contracts written in Solidity and are freely available. Further, they are all automated and require no annotations from the programmer. We selected Oyente, Securify, Mythril, and Manticore as they were used in many smart contract analysis studies [16, 118, 123, 154]. Oyente, Mythril, and Manticore employ symbolic execution, Securify uses static information flow analysis, and Slither uses Static Single Assignment (SSA) for analysis. We included Smartcheck as it uses a pattern-matching approach.

4.3 Motivation and Challenges

This section presents motivating examples of undetected security bugs by static analysis tools, followed by an overview of the challenges in evaluating the tools.

[2]Wei is the smallest denomination of the Ether cryptocurrency in Ethereum.

```
11  uint _vtime = block.timestamp;
12  if (startTime + (5 * 1 days) == _vtime){
```

Figure 4.2: Modification made to the contract in Figure 4.1

4.3.1 Motivating Examples

The contract example in Figure 4.1 has at least two vulnerabilities, (1) two instances of timestamp dependency bug at lines 11 and 19, and (2) one instance of transaction ordering dependence (TOD) represented by the functions `play` and `getReward`, specifically lines 20 and 27 in each function, respectively. The timestamp dependency security bug is that the block's timestamp should not be used in the code as it can be controlled by miners, while the TOD security bug is that the state of the smart contract should not be relied upon by the developer (Section 4.6).

We have used four of the static analysis tools in Section 4.2 (supposed to detect these security bugs) to check this contract for security bugs, Oyente, Securify, Mythril, and SmartCheck. According to the tools' research papers [103, 108, 149, 154], Oyente, Mythril, and SmartCheck should detect the timestamp dependency bug. However, we found that while Oyente and Mythrill were not able to detect both instances of timestamp dependency security bug in lines 11 and 19, Smartcheck detected only the instance in line 19. For the second instance, Smartcheck gave a hint that block.timestamp should be "used only in equalities". To further test SmartCheck's ability to detect the security bug, we made a small modification to the syntax of the smart contract while keeping its semantics the same (Figure 4.2). SmartCheck subsequently failed to detect the bug altogether.

Regarding the TOD security bug, both Oyente and Securify are supposed to detect this class of security bugs. However, we found that only Securify detected this bug successfully, while Oyente was not able to detect it. We extracted the code snippet representing TOD from this contract (lines 15 to 29), injected it in another larger contract free of security bugs, and obtained similar results.

These examples motivated us to prepare multiple code snippets for the different security bugs (within the scope of the tools) and to manually inject them into the code of five smart contracts (the first five contracts in the set of contracts in Section 4.6). We then used the tools to check the buggy contracts, and found sev-

eral instances of undetected security bugs, even though the tools were supposed to detect them. However, it was tedious and error-prone to manually inject these security bugs and inspect the results, so we decided to automate this process.

4.3.2 Automated Security Bug Injection Challenges

The simplest way to inject security bugs into smart contracts is to inject them at random locations - this is how traditional fault injection (i.e., mutation testing) works. However, random injection is not a cost-effective approach as we have to follow specific guidelines for the injected security bug to be exploitable. We identify two main challenges.

Security Bug Injection Locations

As the underlying techniques used by some tools (e.g., symbolic execution) depend on the control and data flow in the analyzed contracts, injecting an instance of each security bug at a single location would not be sufficient. Therefore, security bugs should be injected into all potential locations in the contract code. On the other hand, the process of identifying the potential locations depends on the code of the original contract, and also on the type and nature of each security bug. Injecting security bugs at the wrong locations would result in compilation errors. In addition, it might yield instances of dead code in the contract. For example, injecting a security bug formulated as a stand-alone function inside the body of another function would result in a compilation error, as Solidity does not support nested functions. Moreover, a security bug injected into an 'if' statement condition that would make the condition always fail would make the 'then' clause unreachable.

Semantics Dependency

For the injected security bug to be an active bug that can be exploited by an attacker, it has to be aligned with the semantics of the original contract. For example, assume we want to inject a Denial of Service (DoS) security bug by calling an external contract. We can use an if-statement with a condition containing a call to another contract function. However, for this security bug to be executed, we also need to define the appropriate external contract.

SolidiFI addresses the first challenge by parsing the Solidity code into an Abstract Syntax Tree (AST) and injecting security bugs into *all* syntactically valid locations. It addresses the second challenge by formulating exploitable code snippets for each security bug type.

4.4 *SolidiFI* Approach and Workflow

The main goal of *SolidiFI* is to perform a systematic evaluation of *static analysis* tools used to check smart contracts for known security bugs. Figure 4.3 shows the workflow of *SolidiFI*. The code snippets representing a specific security bug are injected in each smart contract's source code at *all* possible locations (step 1). The selection of the injection locations is a function of the security bug to be injected. SolidiFI *injects security bugs into the source code to imitate the introduction of security bugs by developers. However, its use is not restricted to tools that perform analysis at the source code level. For example, tools that work on the EVM bytecode would compile the buggy contracts to produce the EVM code for analysis.* Then, the injected code is scanned using the static analysis tools (step 2). Finally, the results of each tool are checked, and false negatives and false positives are measured (step 3).

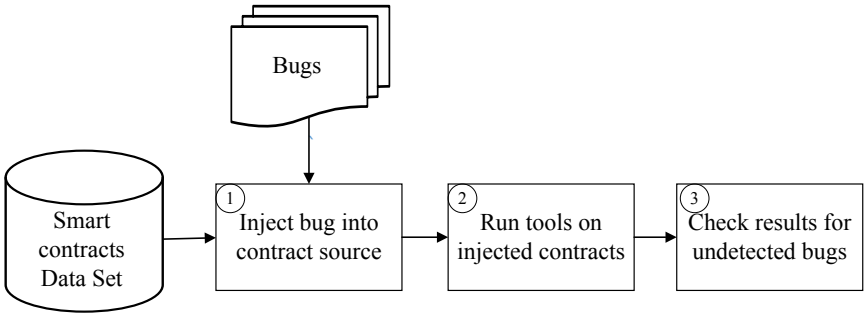

Figure 4.3: *SolidiFI* Workflow.

4.4.1 Bug Model

In our work, a security bug is expressed as a code snippet, which leads to a vulnerability that the security tool being analyzed aims to detect. *SolidiFI* reads code

snippets to be injected from a pre-defined bug pool prepared by us (the bug pool can be easily extended by users to add new bugs). For each tool, we only inject the security bugs that the tool claims to detect. based on the tool's research paper. However, because the tools are continuously evolving, the research paper may not have the up-to-date list of security bugs detected by the tool, and hence we use the tool's online documentation to augment it.

4.4.2 Security Bug Injection

In this work, the security bugs are injected into the source code in three ways as follows.

Full Code Snippet

In this approach, we prepare several code snippets for each security bug under study. Each code snippet is a piece of code that introduces the security bug.

To illustrate the process, we discuss the bug types and example code snippets.

(1) Timestamp Dependency: The current timestamp of the block can be used by contracts to trigger some time-dependent events. Given the decentralized nature of Ethereum, miners can change the timestamp (to some extent). Malicious miners can use this capability and change the timestamp to favor themselves. This security bug was exploited in the GoverMental Ponzi scheme attack [9]. Therefore, developers should not rely on the precision of the block's timestamp. Figure 4.4 shows an example of a code snippet that represents the bug (`block.timestamp` returns the block's timestamp).

```
1 function bug_tmstmp() public returns(bool) {
2     return block.timestamp >= 1546300;
3  }
```

Figure 4.4: Timestamp dependency example.

(2) Unhandled Exceptions: In Ethereum, contracts can call each other, and send Ether to each other, e.g., using `send()`, `call()`, and `transfer()` high-level

functions. If an exception is thrown by the callee contact (e.g., limited gas for execution), the contract is terminated, its state is reverted, and *false* is returned to the caller contract. Therefore, unchecked returned values within the caller contract could be used to attack the contract, leading to undesired behavior. A serious version of this security bug occurred in the "King of the Ether" [9]. Figure 4.5 shows an example (the `send()` instruction requires its return value to be checked for exceptions to make it secure).

```
1 function unhandledsend() public {
2     callee.send(5 ether);
3   }
```

Figure 4.5: Unhandled exceptions example.

(3) Integer Overflow/Underflow: In Solidity, storing a value in an integer variable bigger or smaller than its limits lead to integer overflow or underflow. This can be used by attackers to fraudulently siphon off funds. For example, Figure 4.6 shows an example code snippet in which an attacker can reset the *lockTime* for a user by calling the function *incrLockTime* and passing 256 as an argument - this would cause an overflow, and end up setting the lockTime to 0. Batch Transfer Overflow is a real-world example [114].

```
1 function incrLockTime(uint _sec) public{
2     lockTime[msg.sender] += _sec;
3   }
```

Figure 4.6: Integer overflow/underflow example.

(4) Use of tx.origin: In a chain of calls, when contracts call functions of each other, the use of `tx.origin` (that returns the first caller that originally sent the call) for authentication instead of `msg.sender` (that returns the immediate caller) can lead to phishing-like attacks [135]. Figure 4.7 shows an example snippet in which `tx.origin` is used to withdraw money.

```
1 function bug_txorigin(address _recipient) public {
2     require(tx.origin == owner);
3    _recipient.transfer(this.balance);
4  }
```

Figure 4.7: tx.origin authentication example.

(5) Reentrancy: Contracts expose external calls in their interface. These external calls can be hijacked by attackers to call a function within the contract itself several times, thereby performing unexpected operations within the contract itself. For example, the external call in Line 3 of the snippet code shown in Figure 4.8 can be used by an attacker to call the *bug_reEntrancy()* function repeatedly, potentially leading to withdrawal of Ether more than the balance of the user. The DAO attack [40] is a well-known example exploiting this security bug.

```
1 function bug_reEntrancy(uint256 _Amt) public {
2     require(balances[msg.sender] >= _Amt);
3     require(msg.sender.call.value(_Amt));
4     balances[msg.sender] -= _Amt;
5  }
```

Figure 4.8: Re-entrancy example.

(6) Unchecked Send: Unauthorized Ether transfer, such as non-zero sends, can be called by external users if they are visible to the public, even if they do not have the correct credentials. This means unauthorized users can call such functions and transfer Ether from the vulnerable contract [135]. An example code snippet is shown in Figure 4.9.

```
1 function bug_unchkSend() payable public{
2     msg.sender.transfer(1 ether);
3  }
```

Figure 4.9: Unchecked send example.

(7) Transaction Ordering Dependence (TOD): Changing the order of the transactions in a single block that has multiple calls to the contract, results in changing the final output [6]. Malicious miners can benefit from this. An example code snippet vulnerable to this security bug is shown in Figure 4.10. In this example, the attackers can send a puzzle solving reward to themselves instead of the winner of the game by executing *getReward_tod()* before *setWinner_tod()*.

```
address payable winner_tod;

function setWinner_tod() public {
   winner_tod = msg.sender;
 }

function getReward_tod() payable public{
   winner_tod.transfer(msg.value);
 }
```

Figure 4.10: TOD example.

Code Transformation

This approach aims to transform a piece of code without changing its functionality but make it vulnerable to a specific security bug. We leverage known patterns of vulnerable code to inject this bug. We use this approach to inject two security bug classes that are compatible with this approach, namely (1) integer overflow/underflow and (2) use of tx.origin.

Table 4.1: Code transformation patterns.

Bug Type	Original Code Patterns	New Code Patterns
tx.origin	msg.sender==owner	tx.origin==owner
Overflow	bytes32	bytes8
Overflow	uint256	uint8

Table 4.1 shows examples of the code patterns that are replaced to introduce the security bugs, and the vulnerable patterns for each security bug type.

Figure 4.11 shows an example before and after security bug injection using this

```
1  /*(Before)*/
2  function sendto(address payable receiver, uint amount) public
       {
3     require (msg.sender == owner);
4     receiver.transfer(amount);
5   }
6
7  /*(After injection)*/
8  function sendto(address payable receiver, uint amount) public {
9     require (tx.origin == owner);
10    receiver.transfer(amount);
11  }
```

Figure 4.11: Code transformation example.

approach. In this example, `transfer` instruction is used to perform a transfer of the specified Ether amount to the receiver's account after verifying the direct caller of `sendto()` to be the owner. To inject the tx.origin bug, the authorization condition `msg.sender == owner` should be replaced with the `tx.origin == owner`, in which the owner is not the direct caller of `sendto()`. However, the authorization check is passed successfully, which enables attackers to authorize themselves, and send Ether from the contract, even if they are not the owner.

Weakening Security Mechanisms

In this approach, we weaken the security protection mechanisms in the smart contract code, which protect external calls. Note that our goal is to evaluate the static analysis tool, and not the smart contract itself. We use this approach to inject Unhandled exception security bugs. Figure 4.12 shows an example, in which the *Unhandled exceptions* security bug is injected by removing the `revert()` statement that reverts the state of the contract if the transfer transaction failed - this causes the balance to incorrectly become zero even if the transaction failed.

4.5 *SolidiFI* Algorithm

The process for injecting security bugs takes as input the Abstract Syntax Tree (AST) of the smart contract, and it has the following steps.

```
/*(Before)*/
function withdrawBal () public{
    Balances[msg.sender] = 0;
    if(!msg.sender.send(Balances[msg.sender])) {
        revert();
     }
 }

/*(After injection)*/
function withdrawBal () public{
    Balances[msg.sender] = 0;
    if(!msg.sender.send(Balances[msg.sender])) {
      //revert();
    }
 }
```

Figure 4.12: Weakening security example.

1. Identify the potential locations for injecting security bugs and generate an annotated AST marking all identified locations.

2. Inject security bugs into all marked locations to generate the buggy contract.

3. Check the buggy contract using the evaluated tools and inspect the results for undetected security bugs and false alarms.

We discuss the steps in detail below.

Security Bug Locations Identification: The AST is passed to *Bug Locations Identifier*, that drives a bug injection profile (BIP) of all possible injection locations in the target contract for a given security bug. The BIP is derived using AST-based analysis for identifying potential injection locations in smart contract code by Algorithm 1. Algorithm 1 takes as input the AST and the security bug type to be injected, and outputs the BIP.

To address the first challenge of identifying security bug injection locations as mentioned in Section 4.3.2, we define rules that specify the relation between the security bug to be injected and the target contract structure. In general, bugs take two forms: an individual statement, and a block of statements. A block of statements can be defined either as a stand-alone function, or a non-function block

Algorithm 1: Identifying Injection Locations Algorithm

```
   Output: BIP ▷ Bug injection profile
1  Procedure FindAllPotentialLocations(AST, bugType)
2      foreach Snippet s ∈ Snippets & Snippets ∈ bugType do
3          if snippetForm == simpleStatement then
4              BIP ← walkAST(simpleStatement)
5          else if snippetForm == nonFunctionBlock then
6              BIP ← walkAST(nonFunctionBlock)
7          else if snippetForm == functionDefinition then
8              BIP ← walkAST(functionDefinition)
9      BIP ← BIP ∪ findRelatedSecurityMechanisms()
10     BIP ← BIP ∪ findCodeToTransform()
11     return BIP
```

such as an 'if' statement. Therefore, we use a rule for each form of the security bug that defines the specifications of the locations for injecting it.

To identify such locations, for each distinct form of the code snippets defining the security bug type to be injected, we walk the AST based on the code snippet form and the related rule (lines 2-10 in Algorithm 1). $WalkAST(simpleStatement)$, for example, will visit (parse) the AST and find all the locations where a simple statement can be injected without invalidating the compilation state of the contract, and the same for the other forms of the code snippets of the security bug type. After identifying the locations for injecting code snippets of security bugs, we also look for existing security mechanisms to be weakened to introduce the related security bug, and the code patterns to be transformed for introducing the security bug (lines 9 and 10).

Security Bug Injection and Code Transformation: *SolidiFI* uses a systematic approach to inject security bugs into the potential locations in the target contract. The *Bug Injector* model seeds a security bug for each location specified in the BIP. It uses text-based code transformation to modify the code where the information derived from the AST is used to modify the code to inject security bugs. Three different approaches are used to inject security bugs as discussed in Section 4.4.2. In addition to injecting security bugs in the target contract, *Bug Injector* generates

a *BugLog* that specifies a unique identifier for each injected security bug, and the corresponding location(s) in the target contract where it has been injected.

Buggy Code Check and Results Inspection: The resulting buggy contract is passed to the *Tool Evaluator* that checks the buggy code using the tools under evaluation. It then scans the results generated by the tools looking for the security bugs that were injected but undetected, with the help of *BugLog* generated by the *Bug Injector*. *SolidiFI* only considers the injected security bugs that are undetected. So if an evaluated tool reported bugs in locations other than where bugs have been injected, *SolidiFI* does not consider them in its output of false negatives. This is to avoid potential vulnerabilities in the original contract from being reported by *SolidiFI*, which would skew the results. Moreover, *SolidiFI* inspects results generated by the tools looking for other reported bugs and checks if they are true security bugs or false alarms (more details in Section 4.6).

4.5.1 Implementation

SolidiFI approach is fully-automated (except for the pre-prepared buggy snippets). This involves compiling the code, injecting and generating buggy contracts, running the evaluated tools on the buggy contracts, and inspecting reports of the evaluated tools for false-negatives, misidentified cases, and false-positives (except for the manual validation of the filtered false-positives). To make *SolidiFI* reusable, we did not hard-code the patterns that are replaced to introduce security bugs, but rather made them configurable from an external file. We have made SolidiFI *code publicly available [65]*.

SolidiFI uses the Solidity compiler **solc** (supports compiler versions up to 0.5.12) to compile the source code of the contract to make sure it is free from compilation errors before security bugs are injected. In addition, *SolidiFI* uses **solc** to generate the AST of the original code in JavaScript Object Notation (JSON) format. We have implemented the other components of *SolidiFI* in Python in about 1500 lines of code. These components are responsible for identifying the potential locations for injection, injecting security bugs using a suitable approach, generating the buggy contract, inspecting the results of the evaluated tools, and then reporting the

undetected bugs and false alarms. Finally, we developed a Python client to interact with contracts deployed on the Ethereum network - this client is used for assessing the exploitability of the injected bugs in the generated buggy contracts.

4.6 Evaluation

Our evaluation aims to measure the efficacy of *SolidiFI* in evaluating smart contract static analysis tools, and finding cases of undetected security bugs (i.e., false negatives) and false positives. We also measure the performance of *SolidiFI* itself, as well as the ability to exploit the undetected security bugs. We made all the experimental artifacts used in this study and our results publicly available [66]. Our evaluation experiments are thus derived to answer the following research questions:

RQ1. What are the false negatives of the tools being evaluated?

RQ2. What are the false positives of the tools being evaluated?

RQ3. Can the injected security bugs in the contracts be activated (i.e., exploited) at runtime by an external attacker?

RQ4. What is the performance of *SolidiFI*?

As mentioned earlier, we have selected six static analysis tools for evaluation, Oyente [103], Securify [154], SmartCheck [149], Mythril [108], Manticore [106], and Slither [56]. We downloaded these tools from their respective online repositories, which are mentioned in the corresponding papers (Oyente 0.2.7 [102], Securify v1.0 [153] as downloaded and installed in Dec 2019, Mythril 0.21.20 [107], Smartcheck 2.0 [148], Manticore 0.3.2.1 [105], and Slither 0.6.9 [152]).

To perform our experiments, we used a data set of 50 smart contracts, chosen from the list of verified smart contracts available on *Etherscan* [52], a public repository of smart contracts written in Solidity for Ethereum. We selected these contracts based on three factors namely (1) code size (we selected contracts with different sizes that were representative of Etherscan contracts ranging from small contracts with tens of lines of code to large contracts with hundreds of lines of code), (2) compatibility with Solidity version 0.5.12 (at the time of writing, *312 out of 500 verified smart contracts* in EtherScan supported Solidity 0.5x and higher), and (3) contracts with a wide range of functionality (e.g., tokens, wallets, games). Table 4.2 shows the number of lines of code (including comments), and the number

Table 4.2: Contracts benchmark. F represents Functions, and M represents Function Modifiers

Id	Lines	F+M	Id	Lines	F+M	Id	Lines	F+M
1	103	6	18	406	29	35	317	29
2	128	9	19	218	32	36	383	20
3	132	10	20	308	27	37	368	24
4	117	6	21	353	18	38	195	24
5	250	17	22	383	19	39	52	4
6	161	22	23	308	20	40	465	22
7	165	22	24	741	27	41	160	8
8	251	17	25	196	12	42	128	16
9	249	19	26	143	20	43	285	22
10	39	5	27	336	33	44	298	24
11	193	19	28	195	24	45	156	14
12	281	27	29	312	13	46	125	6
13	161	8	30	711	57	47	223	18
14	185	20	31	216	12	48	232	19
15	160	8	32	143	14	49	52	4
16	248	27	33	129	16	50	171	18
17	128	17	34	445	29			
Average values							**242**	**18**

of functions and function modifiers for each contract.[3] The contracts range from 39 to 741 lines of code (loc), with an average of 242 loc.

We limited ourselves to 50 contracts due to the time and effort needed to analyze the contracts by the evaluated tools and inspect the analysis results of the tools to verify false-positives. *With that said, even with this dataset, SolidiFI found significant numbers of undetected security bugs in the tools (e.g., false negatives), as will be discussed in the following sections.*

As explained in Section 4.4, in our experiments, we injected security bugs belonging to seven different security bug types within the detection scope of the selected tools. Table 4.3 shows the security bug types, and the tools that are de-

[3]Function modifier checks a condition before the execution of the function.

signed to detect each bug type. We chose these security bug types based on the bug types detected by the individual tools, and because these security bugs are common in smart contracts, and lead to vulnerabilities that have been exploited in practice [9, 40, 120]. However, *SolidiFI* is not confined to these bug types.

In our experiments, we set the time-out value for each tool to 15 minutes per smart contract and bug type. If a tool's execution exceeds this timeout value, we terminate it and consider the bugs found as its output. While 15 minutes may seem high, our goal is to give each tool as much leeway as possible. Only two of the tools exceeded this time limit in some cases (i.e., Mythril and Manticore). For these two tools, we experimented with larger timeout values, but they did not significantly increase their detection coverage. Note that the total time taken to run the experiments was already quite high with this timeout value - for example, it took us about 4 days to analyze the contracts using Mythril (50-contracts * 6 bug-types * 15-minutes = 75 hours).

Table 4.3: Security bug types used in our evaluation experiments: '*' means that the tool can detect the bug type.

Bug Type	Oyente	Securify	Mythril	SmartCheck	Manticore	Slither
Re-entrancy	*	*	*	*	*	*
Timestamp dependency	*		*	*		*
Unchecked send		*	*			
Unhandled exceptions	*	*	*	*		*
TOD	*	*				
Integer overflow/underflow	*		*	*	*	
Use of tx.origin			*	*		*

4.6.1 RQ1: What Are the False Negatives of the Evaluated Tools?

The core part of our evaluation is to use *SolidiFI* to inject security bugs and evaluate the effectiveness of the tools in detecting the injected security bugs. We performed the following steps in our experiments. First, *SolidiFI* is used to inject security

bugs of each bug type in the code of the 50 smart contracts, one bug type at a time. The resulting buggy contracts are then checked using the static analysis tools. Finally, the number of injected security bugs that were not detected by each tool was recorded.

To get meaningful results, we inject security bugs that are as distinct as possible by preparing a diverse set of distinct code snippets with different data inputs and function calls- this resulted in 9369 distinct security bugs. We consider two injected bugs as distinct if the static analysis tool under study would reason about them differently based on the underlying methodology, where either the data and control flow leading to the injected security bug is different, or the design patterns of the bug snippets are different. *To ensure a fair evaluation, we inject only the security bugs that are supposed to be detected by each tool.*

We consider an injected security bug as being correctly detected by a tool if and only if it identified both the line of code in which the security bug was injected, as well as the bug type (e.g., Re-entrancy). In many cases, we observed that the tool would correctly identify the line of code in which the bug occurred, but would misidentify the bug type. Therefore, we also report the former separately.

The results of injecting security bugs of each bug type, and testing them using the six tools are summarized in Table 4.4. In the table, "Injected bugs" column specifies the total number of injected security bugs for each bug type, "✓" means we did not find any undetected security bug of that bug type (row), while "NA" means the bug type is out of scope of the tool, i.e., it is not designed to detect the security bug type. The numbers for each column specify the total number of security bugs that were either incorrectly detected or not detected by the tool corresponding to the bug type specified in that row. The number within parentheses specifies the number of cases that were not reported by the tool - this does not consider the incorrect reporting of the bug type. The last row of the table shows the overall detection rate for each tool for the bug classes supported by the tool (which is a fraction of security bugs detected by the tool of all injected security bugs). Note that, the detection rate is only calculated for the injected bugs, and we do not include other cases reported by the tools that are not within those injected. This is because we do not have a ground truth of other true security bugs in the contracts before injection.

Table 4.4: False negatives for each tool. Numbers within parentheses are security bugs with incorrect line numbers or unreported.

Security bug	Injected bugs	Oyente	Securify	Mythril	SmartCheck	Manticore	Slither
Re-entrancy	1343	1008 (844)	232 (232)	1085 (805)	1343 (106)	1250 (1108)	✓
Timestamp dep	1381	1381 (886)	NA	810 (810)	902 (341)	NA	537 (1)
Unchecked-send	1266	NA	499 (449)	389 (389)	NA	NA	NA
Unhandled exp	1374	1052 (918)	673 (571)	756 (756)	1325 (1170)	NA	457 (128)
TOD	1336	1199 (1199)	263 (263)	NA	NA	NA	NA
Integer flow	1333	898 (898)	NA	1069 (932)	1072 (1072)	1196 (1127)	NA
tx.origin	1336	NA	NA	445 (445)	1239 (1120)	NA	✓
Detection Rate%		16.7	69.6	43.3	13	8.6	81.7

From the table, we can see that a significant number of false negatives occur for all the evaluated tools, and that *none of the tools was able to detect all the injected security bugs correctly even if we accepted an incorrect bug type with the correct line number as a detected bug*. The highest detection rate is 81.7% for Slither, and the second best detection rate is achieved by Mythril (69.6%). The detection rates for the other tools fall below 44%. In fact, the only tool that had 100% coverage for individual bug types was Slither, for *Reentrancy* and *tx.origin* bugs. Of all tools, Slither had the lowest false-negatives, followed by Securify across bug types.

Our results thus show that all static analysis tools have many corner cases of security bugs that they are not able to detect. *Note that it is surprising that our technique found as many undetected security bugs by the tools as it did, given that our goal was not specifically to exercise corner cases of the tools in question.* We discuss the reasons for the missed detections and the implications later (Section 4.7).

4.6.2 RQ2: What Are the False Positives of the Evaluated Tools?

A false-positive occurs when a tool reports a bug, but there was no bug in reality. Unlike false negatives, where we know exactly where the security bugs have been injected, and hence have ground truth, measuring false positives is challenging due to the lack of ground truth. This is because we cannot assume that the smart contracts used are free of security bugs (though they are chosen from the verified contracts on Etherscan). Further, manually inspecting each bug report and the related contract involves a tremendous amount of effort due to the large number of bug reports, and is hence not practical.

To keep the problem of determining false-positives tractable, we came up with the following approach. The main idea is to manually examine only those security bugs that are *not* reported by the majority of the other tools for each smart contract. In other words, we conservatively assume that a security bug that is reported by a majority of the tools cannot be a false-positive. However, at worst, we will underestimate the number of false-positives in this approach, subject to the vagaries of the manual inspection process. We also verified that many of the security bugs that are excluded by the majority are indeed false-positives by manually inspecting a random sample of these security bugs.

Even after this filtering, we had to manually inspect a significant number of security bugs to determine if they were false positives. Therefore, for each tool, we randomly selected 20 security bugs of each bug type category that were not excluded by the majority approach, and inspected them manually. For those cases where the number of security bugs is less than or equal to 20, we inspected them all. Based on the results of our manual inspection, we estimated the false positives as the percentage of security bugs inspected that were indeed false positives, multiplied by the number of security bugs filtered (i.e., not excluded).

For example, assume that the total number of security bugs reported by a tool is 100. Of these 100 bugs, let us assume that 60 are also reported by the majority of the other tools for the smart contract, and hence we exclude them. Of the remaining 40 filtered security bugs, we manually examine 20 bugs chosen at random. Assume that 16 of these are indeed false-positives. We assume that 80% of the filtered bugs are false-positives, and estimate the number of false-positives to be 32.

Table 4.5: False positives reported by each tool. Empty cells mean that the tool was not designed for that particular bug type. #N= Reported bugs. FL = Filtered bugs. TP = True Positive. FP = False Positive.

Bug Type	**Threshold**	**Oyente**			**Securify**			**Mythril**			**SmartCheck**			**Manticore**			**Slither**		
=		#N	FL	FP	#N	FL	FP	#N	FL	FP	#N	FL	FP	#N	FL	FP	#N	FL	FP
Re-entrancy	4	0	0	-	12	12	12	54	54	43	0	0	-	6	6	6	79	79	71
Timestamp dep	3	0	0	-				12	12	0	0	0	-				12	12	0
Unchecked send	2				7	4	4	14	3	3									
Unhandled exp	3	10	10	10	0	0	-	0	0	-	6	6	6				0	0	-
TOD	2	32	24	24	121	97	97												
Integer flow	3	947	943	801				17	3	3	3	2	2	9	9	9			
tx.origin	2							0	0	-	3	1	0				4	2	0
Miscellaneous		0			318			144			1520			169			1807		

Table 4.5 summarizes the results of false positives reported by each tool. In the table, the "Threshold" column refers to the majority threshold, which is the number of tools that must detect the security bug in order for it to be excluded from consideration - this number depends on how many tools are able to detect the bug type.

For each tool, the sub-column "#N" shows the number of security bugs reported by the tool, the sub-column "FL" shows the number of security bugs that have been filtered (not excluded) by the majority approach, while the sub-column "FP" shows the false positives of the tool based on the manual inspection. Empty cells in the table represent cases where a tool was not designed to detect a bug type. Note that some of the tools detected bugs outside the seven categories that we considered - we called these as *miscellaneous*.

From the table, we can see that all the evaluated tools have reported a number of false positives, ranging from 2 to 801 for most of the bug types. Interestingly, the results show that the tools with low numbers of false negatives reported high false positives, i.e., *Slither* and *Securify*. For example, although *Slither* was the only tool that successfully detected all the injected Re-entrancy security bugs, it reported significant false positives. This raises the question of whether the high detection

rate was simply a result of over-zealously reporting security bugs by the tool (this is also borne out by the high number of bugs reported under the *miscellaneous* category by this tool). *This highlights the need for security analysis tools that are able to detect security bugs while maintaining low false positive rates.*

We provide some examples of the false-positive cases below. For example, most unhandled exception security bugs were reported even though the code checks the return values of the send functions for exceptions using *require()*. As another example, many false positives were re-entrancy security bugs where the code contains the required checks of the contract balance, and updates the contract states before the Ether transfer. Oyente reported several cases as an integer over/underflow even though they are no integer-related calculations (e.g., *string public symbol = "CRE";*). On the other side, we tried to be consistent with the assumptions considered by the tools during our manual inspection. For example, some of the cases that we considered as true security bugs were re-entrancy security bugs that use the *transfer* function. This function protects against re-entrancy issues as it has limited gas; however, we considered them as true security bugs as the attack can happen if the gas price changes - this is detected by some of the tools (e.g., Slither).

4.6.3 RQ3: Can the Injected Security Bugs Be Activated by an External Attacker?

The goal of this RQ is to assess whether the undetected security bugs in RQ1 can be activated in the contract at runtime. This is to determine whether the reason behind the security bug not being detected by the evaluated tool was that the security bug cannot be activated (and hence cannot be exploited by an attacker). We deploy the set of buggy contracts with the undetected bugs (found in RQ1) on the Ethereum blockchain, and execute transactions that attempt to activate them.

To conduct these experiments, we use MetaMask [36], a browser extension that allows us to connect to an Ethereum node called INFURA [35], and run our buggy contracts on this node. We have created Ethereum accounts on Ethereum Kovan Testnet (test network) using MetaMask, and deposited a sufficient amount of Ether to these accounts to enable us to execute transactions (pay the required gas for transactions). We use Remix [128] (Solidity editor) to deploy contracts on

Ethereum Kovan Testnet. Remix enables us to connect with MetaMask to deploy contracts on the INFURA Ethereum node.

We illustrate the process with an example. As mentioned in RQ1, Manticore did not report instances of injected integer overflow/underflow security bugs - an example is shown in Figure 4.13. Our goal is to attempt to activate this security bug in the deployed buggy contract by calling the function *bug_intou3()*. The returned result was 246 - this is not the expected value (-10) due to the use of an unsigned integer type (i.e., uint8) instead of a signed integer type, which resulted in an integer underflow. Thus, the security bug can be activated by an attacker.

We had to manually craft inputs for each security bug in order to test its activation, which takes significant effort. Because of the large number of undetected security bugs, we selected five undetected security bugs for each bug type randomly from different contracts to test their activation. Table 4.6 shows the results of our activation experiments. In the table "–" means we were not able to perform experiments on this bug type, as it requires the attacker to behave as a miner, which would consume a significant amount of computational resources. The results show that one can exploit (activate) all the selected bugs in their related buggy contracts. Therefore, the infeasibility of activation of the security bug was not the reason that the evaluated tools failed to detect the injected security bugs. Further, the results validate the assumption made in the design of the proposed approach, *SolidiFI*, that *SolidiFI* injects security bugs that can be activated by malicious users. *SolidiFI* tackled the challenge of injecting exploitable security bugs through (1) the identification of the valid injection locations of the security bugs, and (2) the injection of manually prepared code snippets vulnerable to security bugs. With that said, the dependency on manual code snippets may also result in security bugs that cannot be activated. This can happen if the end-users of *SolidiFI* carelessly prepared invulnerable code snippets to be used by *SolidiFI*.

4.6.4 RQ4: What Is the Performance of *SolidiFI*?

Finally, we measured the performance of *SolidiFI* in terms of the time it takes to inject security bugs and generate buggy contracts. We excluded the time of running the tools being evaluated to check the buggy contracts, as this is tool-specific

```
function bug_intou3() public{
    uint8 vundflw =0;
    vundflw = vundflw -10; // underflow
    return vundflw;
}
```

Figure 4.13: Undetected integer underflow bug

Table 4.6: Activity of Selected Undetected Bugs.

Bug type	Selected bugs	Activated bugs
Re-entrancy	5	5
Timestamp dependency	–	–
Unchecked send	5	5
Unhandled exceptions	5	5
TOD	–	–
Integer overflow/underflow	5	5
Use of tx.origin	5	5

and independent of *SolidiFI*. We performed injection of each bug type in each contract five times and calculated the average of the five runs, and then calculated the average of injecting the seven bug types in each contract. The average time of injecting all instances of bug types in a contract is 25 seconds, and the worst case time was 46 seconds (for contract 24, which was the largest contract in our set). *Thus,* SolidiFI *takes less than one minute on average per contract and bug type.*

4.7 Discussion

In this section, we examine the reasons for the false negatives of the tools observed in RQ1. We then examine the implications of the results, and our methodology, on both tool developers and end users. Finally, we examine some of the limitations of *SolidiFI* and threats to validity of our experiments.

4.7.1 Reasons for False-Negatives

To establish a practical understanding of the presented results and why some security bugs were not detected, we will highlight the code snippets for some of the

security bugs that were not detected, and then discuss the reasons behind them. We organize this discussion by tool.

Oyente was not able to detect many instances of injected re-entrnacy, timestamp dependency, unhandled exceptions, integer overflow/underflow, and TOD security bugs as mentioned earlier.[4] According to the paper [103], *Oyente* works on detecting only re-entrancy security bugs that are based on the use of *call.value*. Some of the recent tools, such as *Slither*, consider the detection of re-entrancy security bugs with limited gas that are based on *send* and *transfer*. Those papers claim that *send* and *transfer* do not protect from re-entrancy security bugs in case of gas price change. Furthermore, *Oyente* failed to detect instances of re-entrancy security bugs that are based on the use of *call.value*. One of the TOD code snippets we used in our experiments is mentioned in the running example at Figure 4.1 on lines 15-29, which emulates a simple game and its winner. The malicious behavior occurs when the two transactions are executed in one block and the attacker tries to change the order of the received transactions. To understand why this security bug is not detected by Oyente, in Oyente the EVM bytecode is represented as a control flow graph (CFG). The execution jumps are used as edges that connect the nodes of the graph representing the basic execution blocks in the code. The symbolic execution engine of Oyente uses the CFG to produce a set of symbolic traces (execution paths), each associated with a path constraint and auxiliary data, to verify pre-defined properties (security bugs being detected). Basically, Oyente detects TOD by comparing the different execution paths and the corresponding data flow (Ether flow) for each path. Oyente reports those different execution paths that have different Ether flows.

For Oyente to be able to detect all TOD security bugs successfully, the symbolic execution engine should generate all possible execution paths for the contract to find the erroneous path - this is challenging due to the incompleteness of symbolic execution. It also uses some bounds that limit the symbolic execution.

Smartcheck failed to detect most of the injected security bugs across all the categories. Smartcheck checks for bugs by constructing an Intermediate Repre-

[4]Because the released version of Oyente did not work with the latest version of the Solidity compiler (0.5.12), we made a few changes to the injected contracts to get it to work with Oyente - these did not impact the tool's coverage.

sentation (IR) from the source code, and then using XPath patterns to search for bugs in the IR. This approach lacks accuracy as some security bugs cannot be expressed as XPath expressions. For example, the re-entracy security bug is difficult to express as an XPath pattern and is hence not detected.

Further, because Smartcheck uses XPath patterns that detect specific syntax of some security bugs, even a slight variation in the syntax of the security bug snippets would not match the XPath patterns. For instance, SmartCheck did not report some occurrences of unhandled exceptions. By checking the code snippet for one of the undetected unhandled exception security bugs depicted in Figure 4.14, we found that Smartcheck was not able to detect it as unchecked send because Smartcheck only looks for *send* functions without an if-statement (that checks the return value). However, in this snippet, the send is within an if-statement even though the *revert()* exists in the else clause of the if-statement, which will be triggered on the success of send. The same happens when other functions are used for sending ether (e.g., call, etc.) instead of *send*. This is an inherent problem with syntactic, rule-based tools such as Smartcheck.

```
1 if (!addr.send(42 ether)) {
2     receivers +=1;
3  }
4 else {
5     revert();
6  }
```

Figure 4.14: Unhandled exception code snippet 1.

Mythril was the tool with the largest set of undetected security bugs in our experiments. It failed to detect many instances of re-entrancy, timestamp dependency, unchecked send, unhandled exceptions, integer overflow/underflow, and use of tx.origin. For example, the buggy code in Figure 4.15 was not detected by Mythril. The condition of the if-statement that checks the return value of the send will always be evaluated as true because of the added condition "|| 1==1". Hence, the execution of the contract would be reverted in all cases by the function *revert()*, regardless of whether the send succeeds. *This is incorrect as the execution should only be reverted on the fails of send.* However, Mythril does not detect this

as it only evaluates the *send()* part in the condition of the if-statement rather than evaluating the whole condition with the OR part (|| 1==1).

```
1 if (!addr.send (10 ether) || 1==1) {
2     revert();
3   }
```

Figure 4.15: Unhandled exception code snippet 2.

Mythril is also very slow in term of the time it takes to analyze contracts. Although we set the time-out for analyzing each contract to 15 minutes, as mentioned earlier, we also tried setting the timeout to 30 minutes and did not observe any increase in the number of bugs demonstrating that increasing the time-out has diminishing returns. We also found the number of undetected security bugs increased in the large contracts, as Mythril enumerates symbolic traces and this does not scale well in large contracts.

Like the other tools, Mythril also misreported the types of many of the injected security bugs. Figure 4.16 shows part of a buggy contract injected using *SolidiFI*. The injected contract allows users to manage their tokens and send tokens to each other. We injected a re-entrancy security bug using *SolidiFI* in the contract at lines 185-188. However, Mythril reported the re-entrancy security bug as "Unchecked Call Return Value" (i.e., Unhandled exception) at line 186. By inspecting this line of code, we can see that the return value of the *send* function is checked and the balance is reset to zero on the success of *send*, so there is no unhandled exception as reported. This calls into question Mythril's soundness in detecting this type of security bugs as well as its completeness in detecting Re-entrancy security bugs.

Manticore was not able to detect instances of re-entrancy and integer overflow/underflow. Unlike other evaluated tools employing symbolic executions, we noticed that Manticore takes a long time to analyze smart contracts, and in some cases it times-out. It consumes significant memory space as well on our system. Moreover, Manticore crashed and failed to analyze most of the contracts and threw exception errors. The 50 main contracts used in our experiments consist of 123 analyzable contracts (each contract file may contain more than one contract). Out of them, *Manticore* crashed for 83 contracts injected by re-entrancy bugs and 73 con-

```
177 function transfer(address _to, uint256 _value) public returns (
        bool success) {
178     require(balances[msg.sender] >= _value);
179     balances[msg.sender] -= _value;
180     balances[_to] += _value;
181     emit Transfer(msg.sender, _to, _value);
182     return true;
183   }
184
185 function withdraw_balances_re_ent36 () public {
186     if (msg.sender.send(balances[msg.sender ]))
187         balances[msg.sender] = 0;
188   }
```

Figure 4.16: Part of a buggy contract injected by reentrancy bug.

tracts injected by integer overflow bugs. We reached out to the tool developers to get fixes or explanation for these issues; however, there was no response (as of the time of submission). For those challenges, we evaluated Manticore on detecting only two bug types.

Securify was not able to detect several cases of injected security bugs belonging to re-entrancy, unchecked send, unhandled exceptions, and TOD. In addition, we found many cases where *Securify* failed to analyze the injected contracts and threw an error. Out of the 200 contracts injected by the four bug types (50 contracts for each bug type), *Securify* failed to analyze five contracts injected by unhandled exceptions, four injected by re-entrancy, four injected by unchecked send, and four injected by TOD security bugs. If we excluded the injected security bugs in those contracts, the number of undetected security bugs by *Securify* will be as follows: (re-entrancy: 105, unchecked send: 332, unhandled exceptions: 402, and TOD: 136). *Securify* also reported a high number of TOD false positives compared with *Oyente*. A recent study [57] found that the reported false alarms by Securify are due to over-approximation of the execution.

Slither Although *Slither* has almost 100% accuracy in detecting re-entrancy, timestamp, and tx.origin security bugs, it was not able to detect many instances of unhandled exceptions. Moreover, it had a high number of re-entrancy false positives as mentioned earlier.

4.7.2 Implications

Tool Developers

There are two implications for tool developers. The first implication is that using pattern matching for detecting security bugs, especially by employing simple approaches such as XPaths matching, is not an effective way for detecting smart contract security bugs for the reasons mentioned earlier in this section. The second point is that security bug detection approaches that are based on enumerating symbolic traces are impeded by path explosion and scalability issues. Therefore, there is a need for sophisticated analysis tools that also consider the root causes of the vulnerabilities and semantics of the analyzed code instead of depending only on analyzing the syntax and symbolic traces.

End Users of Tools

For smart contract developers, who are the end users of the static analysis tools, there are three implications. First, they can use *SolidiFI* to assess the efficacy of static analysis tools to choose the most reliable tools with no or low false negatives for their use cases. Second, developers should not rely exclusively on static analysis tools, and should test the developed contracts extensively. *SolidiFI* can help them build test suites by introducing mutations and checking if the test cases can catch them. Finally, the generated security bugs by *SolidiFI* and their relative locations in the code can be used for educating developers on writing secure code.

4.7.3 Limitations of *SolidiFI*

There are three limitations of *SolidiFI*. First, the current version of *SolidiFI* works only on Solidity static analysis tools. Although Solidity is the most common language for writing Ethereum smart contracts and most of the proposed tools target analysis of Solidity contracts, *SolidiFI* functionality can be easily extended to other languages. Secondly, the security bug injection approach employed by *SolidiFI* requires pre-prepared code snippets (for each bug type), which requires some manual effort. However, this is a one-time cost for each bug type (we have provided these as part of the tool). Finally, one of our assumptions in this chapter is that *Solid-*

iFI injects security bugs into the source code to imitate the introduction of security bugs by developers. However, this assumption may not hold in practice, as some of the security bugs introduced by developers may not be the same as those injected by *SolidiFI*. Therefore, the effectiveness of the evaluated tools may not be an exact match to the effectiveness of these tools in detecting security bugs introduced by developers.

4.7.4 Threats to Validity

An external threat to the validity is the limited number of smart contracts considered, namely 50. We have mitigated this threat by considering a wide-range of smart contracts with varying functionality and code sizes. We emphasize that *SolidiFI* covers all syntactic elements of Solidity up to version 0.5.12 and, our dataset contains a wide variety of contracts with different features (e.g., loops). Also, we selected contracts with different sizes that were representative of EtherScan contracts ranging from 39 to 741 locs, with an average of 242 loc (Table 4.2).

There are two internal threats to validity. First, the number of tools considered is limited to 6. However, as mentioned, these represent the common tools used in other studies on smart contract static analysis. Further, they are widely used in both academia and industry. All the tools available are open-source and are being actively maintained (with the exception of Oyente). Further, the implemented prototype of *SolidiFI* is reusable and can be easily extended to evaluate other tools. The second internal threat to validity is that we only injected seven bug types. However, these bug types have been (1) considered by most of the tools evaluated, and (2) exploited in the past by real attacks. Therefore, we believe they are representative of security bugs in smart contracts.

Finally, a construct threat to validity is our measurement of false-negatives and false-positives. For false-negatives, it is possible that the security bugs cannot be exploited in practice. We have partially mitigated this threat by sampling the set of false-negatives and attempting to exploit them (RQ3). For false-positives, it is possible that the reported security bugs are true positives. Again, we have partially mitigated this threat by conservatively considering the bugs reported by the majority of the tools for each bug category as true positives.

4.8 Summary

This chapter proposed *SolidiFI*, a technique for performing a systematic evaluation of Ethereum smart contract's static analysis tools based on security bug injection. *SolidiFI* analyzes the AST (Abstract Syntax Tree) of smart contracts and injects pre-defined security bug patterns at all possible locations in the AST. *SolidiFI* was used to evaluate six smart contract static analysis tools, and the evaluation results show several cases of security bugs that were not detected by the evaluated tools even though those undetected security bugs are within the detection scope of tools. *SolidiFI* thus identifies important gaps in current static analysis tools for smart contracts, and provides a reproducible set of tests for developers of future static analysis tools. It also allows smart contract developers to understand the limitations of existing static analysis tools with respect to detecting security bugs.

The remaining part of this dissertation targets advancing the state of static analysis for smart contracts, considering the drawbacks and limitations of the current static analysis tools. In Chapter 5, we present our first vulnerability detection approach, *eTainter*, for finding gas-related vulnerabilities and code smells.

Chapter 5

Detecting Gas-Related Vulnerabilities and Code Smells in Smart Contracts

The findings of the evaluation study of current static analysis tools, which we have conducted in the previous chapter (Chapter 4), showed that all existing tools have significant false-negatives and false-positives. To address this problem, in this chapter, we propose an effective static analysis approach for finding gas-related vulnerabilities and code smells based on static taint analysis; without relying on pre-defined code patterns and rules. We first discuss the problem of gas-related vulnerabilities and code smells in smart contracts and the principal drawbacks of the current work to address this problem. We then illustrate the challenges of finding gas-related vulnerabilities and code smells through a running example and discuss the four different vulnerability and code smell classes, as well as our intuitive insight to detect them using taint analysis. After that, we present our proposed approach, *eTainter*, for performing an internal-procedural static taint analysis in EVM bytecode. Finally, we show the evaluation results of *eTainter* in finding gas-related vulnerabilities and code smells.

5.1 Introduction

As discussed in Chapter 1 (Section 1.2.2), running out of gas or exceeding the block gas limit during a transaction's execution results in exceptions, abortion of the transaction, and reverting all the changes made by the transaction, thereby leading to unwanted behaviors. Unfortunately, smart contracts may contain code patterns that can be exploited by malicious users due to out-of-gas exceptions. We refer to these as *gas-related vulnerabilities*. Further, smart contracts may also contain code patterns that limit the gas needed for execution, leading to unexpected behavior due to changes in the contracts' gas execution costs, e.g., contracts cannot send and receive Ether. We refer to these as *gas-related code smells*. We consider both gas-related vulnerabilities and code smells in this chapter.

Discovering gas-related vulnerabilities and code smells in smart contracts by developers is not straightforward as the gas cost of a contract is based on (1) the cost of the EVM low-level instructions rather than its source code, (2) the global state of the contract, i.e., data in the contract storage, and (3) the cost and behavior of the other external contracts the contract interacts with, which is not apparent from its code. A recent study [178] found that it is time-consuming to manually identify gas-related vulnerabilities in smart contracts. Therefore, we need automated tools to analyze smart contracts for gas-related vulnerabilities and code smells *before deployment* on the blockchain.

We propose an efficient *static-analysis*-based approach, *eTainter* (EVM Tainter), to find gas-related vulnerabilities and code smells in smart contracts. Our main insight is that gas-related vulnerabilities and code smells are caused by the dependency of contract code on data items that are either provided or manipulated by the contract users. We formulate the detection of gas-related vulnerabilities and code smells as a taint analysis problem [132]. We then use static *taint tracking* to find gas-based vulnerabilities and code smells, without any pre-existing code patterns and rules. The main challenge in static taint analysis for smart contracts is that contracts have certain peculiarities (e.g., taint propagation via contract's storage, multiple entry points, access control), as well as patterns of use (e.g., loops for processing strings and bytes), which must be taken into account to avoid false-positive and false-negative results.

To the best of our knowledge, eTainter *is the first technique to target a broad class of gas-related vulnerabilities and code smells without requiring pre-specified code patterns, via static taint analysis. Our contributions are as follows.*

1. Proposing an inter-procedural static taint-analysis approach for detecting multiple classes of gas-related vulnerabilities and code smells, *eTainter*, which tracks taints through the contract's persistent storage and the flow of taints from the contract's multiple entry points, in addition to using domain-specific optimizations to reduce false-positives.

2. Implementing the proposed approach, *eTainter*, as an automated tool that works on the smart contract's EVM bytecode. Our implementation is publicly available at [69].

3. Evaluating the efficacy of *eTainter* on a custom dataset of 28 smart contracts with hand-annotated gas-related vulnerabilities; a set of 60,612 unique smart contracts deployed on Ethereum; and a set of the 3,000 most frequently used contracts in Ethereum.

The results of evaluating *eTainter* show that *eTainter* is able to find gas-related vulnerabilities and code smells with a precision and recall of over 90%. Our comparison with the prior work, MadMax, on detecting gas-related vulnerabilities shows an F1 score of 92% for *eTainter* compared with 69% for MadMax for the same set of contracts, with both precision and recall being higher for *eTainter*. Further, *eTainter* has an analysis time of about 8 seconds on average per smart contract, and it analyzed successfully 88% of the Ethereum contracts without timing out; 97% of these contracts were analyzed within one minute each by *eTainter*. Finally, we find that gas-related vulnerabilities are present in about 6.6% of the contracts in the Ethereum dataset and in about 4.8% of the most frequently-used contracts. Similarly, gas-related code smells are prevalent in smart contracts (25% and 22% in the Ethereum and most frequently-used contracts datasets, respectively).

5.2 Motivating Example

We use a real-world contract to illustrate challenges of finding gas-related vulnerabilities. We also use this as a running example in Section 5.3 and Section 5.4.

```
1  contract PIPOT {
2   uint public fee = 20;
3   mapping(uint => uint) jackpot;
4   struct order{
5     address player;
6     uint betPrice;}
7   mapping(uint => order[]) orders;
8
9   function buyTicket(uint betPrice)public payable{
10     orders[game].push(order(msg.sender, betPrice));
11     uint distribute = msg.value * fee / 100;
12     jackpot[game] += (msg.value - distribute);
13    }
14   function start(uint winPrice)public onlyOwner(){
15     if (orders[game].length > 0) {
16        pickTheWinner(winPrice);}
17     startGame();
18    }
19   function pickTheWinner(uint winPrice) internal {
20     uint toPlayer = jackpot[game]/orders[game].length;
21     for(uint i=0; i<orders[game].length;i++){
22        if (orders[game][i].betPrice == winPrice){
23           orders[game][i].player.transfer(toPlayer);
24        }
25     }
26    }
27 }
```

Figure 5.1: Example of smart contract with two classes of gas-related vulnerabilities.

The code in Figure 5.1 is simplified from the PIPOT contract on Etherscan [53], an explorer for Ethereum. Due to space constraints, we show only the relevant parts of the code. The contract implements a lottery/game where the users of the contract can participate in the game by buying tickets and guessing a bet price. The winners of the game are the participants who correctly guess the bet price specified by the owner of the contract. Users can participate in the game by calling the function *buyTicket* at line 9. The function *buyTicket* stores the address of each participant and the bet price in the mapping *orders* (line 10), which is a data structure in Solidity similar to a hash table. It also deducts 20% of the bet price as a fee and adds the remaining 80% to the mapping *jackpot* (lines 11-12).

The owner of the contract can start a new game by calling the function *start* (line 14), and specifying the winning price for the current active game (parameter *winPrice*). The current game will be deactivated when the owner starts a new game,

and this leads to the *pickTheWinner* function (line 19) getting called by the *start* function (line 16). The *pickTheWinner* function distributes the jackpot evenly to the winners of the game, which are the participants who correctly guessed the bet price picked by the owner of the contract.

The *pickTheWinner* function sends the jackpot share to each winner by iterating through the mapping *orders* in a for-loop (line 21), which has the two classes of vulnerabilities: (1) Unbounded Loop and (2) DoS with Failed Call, in addition to having the Hardcoded Gas Call code smell. The first vulnerability is that the loop is iterating through the dynamic mapping *orders* that grows over time. When a large number of players participate in the game, the loop fails, as the cost of executing the loop exceeds the block gas limit, and none of the winners will receive the prize money. This also locks out the jackpot money in the contract forever as there are no other functions in the contract to transfer the money out.

The second vulnerability is that the function *pickTheWinner* is sending money for the winners within the loop. If one of the winners refuses receiving the money or fails to receive it by throwing an exception, the whole loop will fail, and none of the winners will receive the jackpot. The jackpot money gets locked out in the contract forever.

Furthermore, the loop has a hardcoded gas call code smell due to the use of *transfer*, which limits the gas for transferring Ether to 2300 gas units. Thus, if any of the recipient contracts has an expensive fallback function that consumes more than 2300 gas units, or if any changes occur to the gas cost of EVM instructions executed by the fallback functions, *transfer* fails, and as a result the whole loop fails, leading to locking out the jackpot money in the contract. Both the vulnerabilities and the code smell have serious consequences, as they destroy the promise of the game and the trust in the contract.

When we analyzed this contract with the MadMax tool [73], we found that it failed to detect either of the vulnerabilities, or the code smell. The code smell detection is out of the scope of MadMax, but both vulnerabilities are within the scope of MadMax's detection capabilities. To understand why the vulnerabilities were not detected by MadMax, we looked into the source code of MadMax (available on Github). We found that to check for unbounded loops, MadMax checks if a loop iterates through an array, bounded by the array's size variable (e.g., *a.length*).

In other words, MadMax checks if any of the variables used in the loop represents the array's size. Concretely, in EVM, the first element of a storage array is used to store the length of the array, and it is located at a *constant address* in the contract's storage. This constant address is used in the bytecode to calculate the addresses of the array elements using a `SHA3` hash instruction. Therefore, MadMax detects that a variable used in the loop's condition represents the array's size, if and only if the variable is used in an `SHA3` instruction. Similarly, to check for DoS with failed call, MadMax checks if the address of a call within a loop is an array element. To do so, it checks if the element is part of an array whose size is used in `SHA3` instruction. Unfortunately, these rules does not work for nested structures (e.g., multi-dimensional arrays) such as the one in this example. This is because the address of the variable used to store the length of a nested structure is a hash address that is resolved at run-time, rather than a constant address. Therefore, the pre-specified rules in MadMax are unable to reason about the structure used in the loop, and hence MadMax does not detect these vulnerabilities.

This example illustrates the challenge of using pre-specified rules to identify gas-related vulnerabilities (as done by MadMax). Even if the rules are modified to handle this particular case, there are many other variations that are difficult to express as pre-specified rules.

5.3 Gas-related Vulnerabilities & Code Smells and Detection Via Taint Analysis

eTainter uses *taint tracking* to find gas-related vulnerabilities and code smells without using predefined rules. In what follows, we discuss the taint sources in our analysis, describe each vulnerability and code smell, and then explain how taint analysis can detect it.

5.3.1 Taint Sources

In our analysis, the taint sources are the EVM instructions that introduce user data. These instructions read either the arguments of the invoked contract's function or the transaction-related data (e.g., sender and Ether value).

The first row of Table 5.1 shows the EVM instructions that are defined as taint

sources. The EVM instructions CALLDATALOAD and CALLDATACOPY read data passed as arguments when calling a contract's function, CALLER and ORIGIN return the sender and the origin of the transaction, respectively, and CALLVALUE returns the Ether value of the transaction. In our running example, the parameter *betPrice* (line 9) of the function *buyTicket* is read by the EVM instruction CALLDATALOAD in the bytecode. The global built-in variable *msg.sender* (used in line 10) corresponds to the EVM instruction CALLER, which returns the address of the contract's caller (transaction sender) in the bytecode.

Further, in the bytecode, the EVM instruction, SLOAD, loads data from the contract's storage. In line 21 of our running example, the length of the array read by *orders[game].length* is loaded from the storage, and this pattern is translated to SLOAD instruction in the bytecode. Our analysis initially considers the data read from a storage slot as a tainted data; thus, the EVM instruction SLOAD is defined as a source of taints. However, because not all data in the storage is actually controlled by the user, our analysis performs further checks to validate if storage sources are controlled by the contract users as discussed in Section 5.5.1.

Table 5.1: EVM instructions defined as sources and sinks by *eTainter*. NA = Not applicable (i.e., No constraints).

Type	EVM Instruction(s)	Additional Constraints
Sources	CALLDATALOAD, CALLDATACOPY, CALLER, ORIGIN, CALLVALUE, SLOAD	NA
Sink	CALL < gas, **addr**, value,...>	(1) CALL is in a loop's body, and (2) CALL's return the condition of a revert
Sink	CALL < gas, **addr**, value,...>	The CALL's return is not a condition of a revert
Sink	CALL < **gas**, addr, value,...>	NA
Sink	JUMPI <destination, **condition**>	NA
Sink	ISZERO <**value**>	NA
Sink	SSTORE < key, **value**>	NA

5.3.2 Gas-Related Vulnerabilities and Code Smells

We follow the definition of gas-related issues from: (1) Ethereum smart contract best practices and known attacks [33][141], (2) Solidity documentation: security

considerations [142], and (3) prior work, MadMax [76]. We define three vulnerability classes and one code smell class. We cast each vulnerability and code smell class as an instance of the taint analysis problem.

Unbounded Loops

This class of vulnerabilities occurs when a loop iteration is determined based on user input. The most common form of this vulnerability are loops that iterate through a dynamic data structure (e.g., array, mapping) that grow over time, and can be manipulated by the contract's users [44, 164], as in line 21 of the example in Figure 5.1.

Another example of this vulnerability class are implicit loops generated due to Solidity's programming patterns. Such loops iterate over all items of dynamic arrays, e.g., the code pattern "*arrayName = new dataType[](0)*" used by developers to clear arrays. Such code pattern, "*creditorAddresses = new address[](0);*" from the GovernMental contract, was behind the vulnerability responsible for locking out about 1100 Ether (worth about $2.5 million) [160].

The code above compiles to a loop that iterates over the elements of the array *creditorAddresses* and sets them to zero, effectively deleting them, as shown by the following pseudocode: **foreach** $a \in$ array A **do** $a \leftarrow 0$; When the number of creditors in *creditorAddresses* is large, the gas cost of executing the loop exceeded the block gas limit. This resulted in the execution of the enclosing function being reverted and prevented it from doing its job.

Detection: Taint analysis can find that a loop is unbounded by defining the condition of the loop as a sink and then checking if the tainting sources (data loaded from storage slots written or manipulated by contract users) reach the defined sink. For the example in Figure 5.1, taint analysis checks if contract users can change the length of the array *orders[game]* used as a bound in the loop (line 21). The condition in this loop ($i < orders[game].length$) is the sink, and *msg.sender* is the source. Taint analysis finds data flows in the *buyTicket* function (line 10) in which the taint source reaches the array *orders[]*, causing an increase in the array size. Loops that iterate over user inputs (e.g., function parameters) are commonly used, and reporting all these loops would result in high false-positives. Therefore, our

analysis only detects the loops that can result in a Denial of Service (DoS) due to being bound by storage data items.

In our analysis, sinks in the EVM bytecode are defined by first locating the basic blocks forming loops' exit conditions. Then the arguments of the logical instructions (e.g., `ISZERO`) along with the condition arguments of conditional jump instructions (e.g., `JUMPI`), used to direct execution of the loop, are defined as sinks. The condition in the example (*i¡ orders[game].length*) forms a basic block with the last instruction as `JUMPI <destination, condition>` that directs the execution to the loop body. The condition argument of the `JUMPI` instruction is defined as a sink.

DoS with Failed Call

This vulnerability occurs when external calls are performed in the body of a loop (e.g., to pay users by sending Ether to several addresses) [165]. In smart contracts, it is not advisable to group external calls in a single transaction, i.e., within a loop, and calls should rather be isolated to their own transaction, e.g., each user should withdraw their own Ether. That is because the common practice is to check that each call succeeds, and revert the execution if not. When grouping transactions, if one of the target call recipients has an error (e.g., with a gas-expensive fallback function), the failed call throws an exception, the whole execution gets reverted, and the loop never completes (i.e., no one gets paid). The loop in Figure 5.1 (line 21) has this vulnerability as *transfer* (line 23) is designed to revert on failures, which an attacker can exploit if they are able to make a malicious contract (e.g., one with an expensive fallback function in terms of gas) a target of the external call executed in the loop's body.

Detection: Taint analysis can be used to detect loops having this vulnerability by defining the target address of a call executed in the loop's body as a sink if the return of the call is the condition of a revert statement in the loop's body. Then taint analysis checks if tainting sources (user-defined data loaded from storage) can reach the sink. In this example, the target address (*orders[][].player*) of the *transfer* statement at line 23 is defined as a sink for the taint analysis. Then, taint analysis finds data flows in the buyTicket function (line 10) in which the tainting source

msg.sender is reaching the array *orders[]*, storing the target address of the transfer. In the EVM bytecode, the external calls are translated into `CALL` instructions of the form `CALL <gas,address,value,..>`. Our analysis defines as sinks the `address` arguments of the `CALL` instructions found in the loop body that are followed by revert instructions.

Unhandled External Calls

This vulnerability arises when the target address of an external call is provided by the contract user, but the contract code does not handle the external call failures (i.e., revert execution on failures). When performing an external call to an untrusted address, the successful execution of the external call cannot be guaranteed. For example, the gas cost of executing the target function may exceed the gas dedicated to the call (e.g., having an expensive fallback function), and as a result, the call may run out of gas and fail. Therefore, if the contract code does not handle failures of the external call, the contract resumes execution even when the call fails. This can lead to inconsistencies in the contract state that may change the intended behavior of the contract, and be exploited by attackers to harm the contract's users (e.g., block the service provided by the contract). There have been many real-world contracts in which this vulnerability was exploited. For instance, the KotET contract [93] is a chain-game contract in which one user is monarch (ruler) of the game. The contract enables any user to claim the throne of the game and become the monarch for a claiming price that goes to the current monarch as compensation. However, not handling the exceptions of the call that transfers the compensation amount to the current monarch resulted in wrongly making a user, who was claiming the throne, the monarch of the contract without the previous monarch receiving any compensation due to failure of the transfer call. This is the vulnerability in the contract code.

Figure 5.2 shows an example of a contract function having this vulnerability [161]. In this contract, the function *withdraw* is used to withdraw Ether from the contract using the *send()* function (line 5), which sends withdrawn Ether to the caller contract. The code first deducts the withdrawn Ether from the *balances* and *etherLeft* state variables (lines 3-4). Then it sends the Ether to the caller's

address (line 5). If the fallback function of the target contract (caller contract) consumes more than the gas included with the *send()* call (2300 gas units), the *send()* call will fail and return *false*. Because the contract does not check the return value of the *send()*, the changes made to the contract state variables will not be reverted, and the contract will end up tracking incorrect balances of the contract users, which could prevent the user from withdrawing their Ether again forever.

```
1 function withdraw(uint256 _amount) public {
2     require(balances[msg.sender] >= _amount);
3     balances[msg.sender] -= _amount;
4     etherLeft -= _amount;
5     msg.sender.send(_amount);
6 }
```

Figure 5.2: Unhandled external call example.

Detection: This type of vulnerabilities can be detected using taint analysis by defining the target address of each external call in the contact as a sink when the return value of the call is not a condition of a revert statement. Then taint analysis checks for taint flows from taint sources to the defined sinks. In Figure 5.2, the external call *send()* at line 5 is the sink. Taint analysis finds that the taint source *msg.sender* that returns the caller address is reaching the target address of the *send()*, defined as a sink. Our analysis targets external calls in which target addresses are controlled by contract users. In these calls, the gas cost of the code executed by calls is not known and can be controlled by attackers. Thus, these calls are subject to failures due to out-of-gas exceptions.

The high-level call functions that do not propagate exceptions, *address.send()* and *address.call()*, are translated by the compiler to `CALL` instructions in the bytecode that have the form `CALL<gas, address, value, ..>`. The address argument for the `CALL` instruction is the target address of the call, and we define this argument as a sink for this vulnerability class.

Hardcoded Gas Calls

In Solidity, developers can use built-in functions to send Ether to other Ethereum accounts, e.g., *transfer()*, *send()*, and *call.value()*. The maximum gas used by *transfer()* and *send()* is set to 2300 units (at the time of writing), and developers can

use *call.value().gas(<gasvalue>)* to limit the gas used by the *call.value()* function. In the bytecode, all the high-level calls (*transfer()*, *send()*, and *call.value()*) are translated to "`CALL <gas, address, value, ..>`". The `gas` argument gets assigned by the Solidity compiler to the gas specified for the call. However, if the gas is not specified, the compiler sets the value of the `gas` argument to the transaction's total available gas returned by the built-in function *gasleft()*.

Historically, after the DAO attack in 2016, Ethereum recommended the use of calls with fixed gas (*transfer* and *send*) to guard against reentrancy attacks, so that the malicious callee contracts will not have enough gas to callback (reenter) the caller contract. However, the gas costs of EVM instructions are subject to change during Ethereum hard forks (Ethereum protocol upgrades), which could break already deployed contracts with such fixed gas assumptions, as the calls might fail due to the increased gas cost of fallback functions influenced by the gas cost changes of EVM instructions [32, 137]. An example is the Istanbul hardfork-EIP-1884 [146], released on September 30, 2019, that broke hundreds of real-world contracts due to an increase of `SLOAD` cost from 200 to 800 [30]. This led to an increase in the gas cost of contracts using `SLOAD` instructions in their fallback functions. As a result, the functionality of the contracts performing harcoded-gas external calls to these contracts (having new cost) was broken (i.e., could not receive Ether from one another) as the calls to transfer funds ran out of gas.

The Istanbul hardfork shed light on the negative aspects of limiting the gas of calls to prevent reentrancy attacks such as the one above. The recommendation by Ethereum and security practitioners [31] since then is to instead use *call.value()*, and to prevent such attacks using the Checks-Effects-Interactions pattern [143].

Detection: In the bytecode, to find calls with pre-specified gas assumptions (i.e., limited gas values), our analysis uses the `gas` arguments of the EVM `CALL` instructions as sinks. Taint analysis then tracks data flows to the defined sinks to find calls in which the values of the gas arguments are defined through taint sources or predefined constant values rather than the transaction's total available gas coming from the return value of `gasleft()`, the built-in function to get the remaining gas in Solidity (`GAS()` in the bytecode).

5.3.3 Vulnerability Protection Mechanisms

In some cases, developers follow specific programming best practices to mitigate the risk of the discussed vulnerabilities and protect the contracts from being exploited [162]. For example, developers may mitigate the risk of an unbounded loop by splitting the loop over multiple transactions safely [164]. Another example is implementing access control to prevent exploiting a vulnerability through a public function by restricting the call of the function to only the owner of the contract. We refer to the code patterns used to implement such mechanisms as protective patterns. Our analysis does not report cases with protective patterns as vulnerabilities (see Section 5.5.2).

5.4 Our approach: eTainter

This section presents *eTainter*, our taint analysis approach for detecting gas-related vulnerabilities and code smells in smart contracts. At a high-level, *eTainter* takes as input the EVM bytecode of the smart contract being analyzed; it builds the Control Flow Graph (CFG) and identifies the EVM instructions that introduce user data in the bytecode (taint sources) and the EVM instructions that can form gas-related vulnerabilities or code smells (sinks). It then extracts the CFG paths leading to the defined sinks and performs taint analysis on these paths. Finally, *eTainter* searches for possible protective patterns and excludes protected vulnerabilities. It reports the found vulnerabilities and code smells to the users, along with the corresponding vulnerable functions causing the vulnerabilities. In the following subsections, we explain our approach using the running example in Figure 5.1.

5.4.1 CFG Construction

eTainter constructs a context-sensitive, inter-procedural CFG, i.e., constructed for the entire contract rather than for individual functions. The CFG is built for the EVM runtime bytecode of the contract, i.e., the code deployed on the Ethereum blockchain, and represents all functions of the contract and the interaction between them. *eTainter* performs an inter-procedural analysis, with any public/external function or fallback function in the CFG used as an entry point for the analysis (Section 4.2).

5.4.2 Extracting Vulnerable Paths

eTainter identifies paths leading to gas-related vulnerabilities and code smells as specified in Algorithm 2. The algorithm takes as input the sinks for each class of gas-related vulnerabilities and code smells (as discussed in Section 5.3). It uses the algorithm proposed by Krupp et al. [96] to compute a backward slice (over the created CFG) for each sink instruction (line 2). This is done to reduce the number of paths analyzed for taint flows to the defined sink. If the slice contains any of the instructions defined as taint sources, *eTainter* traverses the CFG and finds paths leading to the sink under analysis, from which all instructions of the slice can be reached, to perform taint analysis.

Then for each extracted path, starting at the root, the algorithm propagates taints through the stack, memory, and storage based on the semantics of the EVM instructions (line 4). *eTainter* uses *taintAnalysis* function (lines 22-27), which returns a tuple `<tsink,src>`, where `tsink` defines whether the sink being analyzed is tainted in the current path, and `src` is a set of the taint sources reaching the analyzed sink. When taints reaching the sink are loaded from slots in the contract's storage, the algorithm checks if the corresponding storage slots are also tainted (lines 5-16). Finally, *eTainter* checks for protective patterns in lines 17 and 20.

To check if a storage slot is tainted, *eTainter* first searches for storage write (`SSTORE`) instructions in the contract and defines them as sinks (line 7). Then *eTainter* traverses the CFG to find paths leading to the defined storage sink instructions for taint analysis (line 9), and checks for taint flow in each path (line 10) – we discuss this further in Section 5.5.1. If a taint flow is found, the address of the storage slot written to by `SSTORE` is added to list of tainted slots (lines 11-14), and the storage slot is confirmed as tainted. When the taints flow from another storage slot (storage-to-storage taint flow) not marked as tainted, the algorithm repeats the process to check if this new storage-slot source is also tainted (lines 15-16).

To propagate taints, for instructions defined as sources, *propagateTaint* procedure introduces taints based on the semantics of the source instructions. Otherwise, *propagateTaint* propagates taints according to the taint propagation rules (line 25). *eTainter* uses three main taint propagation rules, summarized in Table 5.2:
Rule-1: For instructions that take one or more operands and derive a value, if any

Table 5.2: Taint propagation rules. INS = Instruction.

Rule-1	$V = INS(x_1,...,x_n), \exists x_i \in Tainted \implies V = tainted$
Rule-2	$BlockChain = INS \parallel INS = BlockChain \not\Longrightarrow$ Propagate
Rule-3	$V = INS(m_i/s_i), m_i/s_i \in Tainted \implies V = tainted$

operand is tainted, the derived value is tainted.

Rule-2: For instructions that query the state of blockchain, such as `TIMESTAMP`, no taint propagation happens since blockchain state information (e.g., block numbers, timestamp) is not manipulated by the contract users. Similarly, no taint propagation happens for instructions that write to the blockchain (do not include `SSTORE` instruction that modify contract's storage), such as `LOGn` instructions that archive logs generated by the contract events and `CREATE` instruction that creates other contracts. The changes made by such instructions neither influence the contract execution nor change the contract state (e.g., the logs written to the blockchain are not accessible from the contract code).

Rule-3: For instructions that read their input from memory (e.g., `SHA3`), if the data read from memory is tainted, the result value is tainted. The same holds for loading data from storage using `SLOAD` instruction. *eTainter* considers taintedness of the loaded data rather than addresses. For example, when propagating taints for `V=SLOAD(`s_i`)` instruction that loads data from the storage address s_i, *eTainter* will mark `V` tainted only if the data stored at location s_i is tainted. To do so, *eTainter* propagates taints through the contract storage.

Propagating taints through the contract storage is challenging for two main reasons. First, the data stored in the contract storage is persistent (i.e., maintained over transactions). Second, EVM uses an unconventional method to access arrays and mappings stored in the contract storage. We discuss how we address these challenges next in Section 5.5.1.

5.4.3 Implementation

We have implemented *eTainter* in an automated tool working on the EVM bytecode of smart contracts. *eTainter* generates the bytecode by compiling the input source code using the *solc* Solidity compiler. *eTainter* uses a modified version (modified

Algorithm 2: Extracting vulnerable paths

```
   Input: sources,sinks,cfg
   Output: {Pv,Pt} ▷ Vulnerable & tainting paths
 1 begin
 2     slices ← BackwardSlicing(sinks,cfg)
 3     foreach Path p ∈ ExtrPaths(slices,sinks,cfg) do
 4         <tsink,src> ← TaintAnalysis(p,sources,sink)
 5         if tsink & src ∈ Storage then
 6             tSlots ← {} ▷ list of tainted storage slots
 7             sSinks ← find(sstore)
 8             sSlices ← BackwardSlicing(sSinks,cfg)
 9             foreach path ps ∈ ExtrPaths(sSlices,sSinks,cfg)
                do
10                 <tslot,src> ←
                    TaintAnalysis(ps,sources,sSink)
11                 if tslot & (src ∉ Storage | src ∈ tSlots) then
12                     if src ∉ Storage then
                           ▷ Add address of sstore sink to the list of tainted slots
13                         tSlots ← tSlots ∪ sSink.addr
14                     break
15                 else if src ∈ Storage then
16                     go to 7
17             if tslot & not IsProtected(p,ps) then
                   ▷ No protective patterns
18                 {Pv,Pt} ← {Pv,Pt} ∪ {p,ps}
19         else if tsink then
20             if not IsProtected(p) then
21                 {Pv,Pt} ← {Pv,Pt} ∪ {p,p}
22 Procedure TaintAnalysis(P, Sources, sink)
23     foreach instr ∈ P do
24         if instr ∈ Sources | instr ∉ sink then
25             <tMem′,tStorage′,tStack′> ←
                PropagateTaint(tMem,tStorage,tStack)
26         else if instr ∈ sink & sink ∈ Tainted then
27             return {true, taints-reaching-sink}
```

in this work) of the code used in *teEther* [96], a symbolic analysis framework for smart contracts, to generate the control flow graph (CFG) and compute backward slices. Further, *eTainter* uses *rattle* [74], a framework for recovering the Static Single Assignment (SSA) [39] form of the bytecode as the current implementation of *eTainter* uses SSA form for performing taint analysis to handle data overwrites.

5.5 Design Decisions

In this section, we discuss the design decisions we made while implementing our approach to address challenges of (1) propagating taints through storage and memory, (2) checking for protective patterns, and (3) identifying loops in the EVM bytecode.

5.5.1 Handling Storage and Memory Taints

Unlike in traditional languages, taint analysis in smart contracts has to deal with the contract's storage, used to store persistent data, besides volatile memory, used to store transient data. In this section, we discuss the challenges encountered in propagating taints in storage and memory, and how we address them.

Validating Storage Taints

When the taint analysis module finds a flow of tainted data loaded from the storage to a specific sink, the sink will be flagged along with the storage slot(s) from which the data is loaded. However, considering all data loaded from the storage as tainted data will increase the number of false positive cases, as not all storage slots can be manipulated by the contract's users. For example, in the loop at line 21 of the running example, the taint analysis module finds that the value of "*orders[game].length*" (marked as tainted since it is loaded from the storage) reaches the loop condition "*i<orders[game].length*" defined as a sink. *eTainter* needs to check whether users of the contract can indeed change the value of "*orders[game].length*" to mark it as tainted, as only then can the array grow over time, and the vulnerability be exploited.

As specified in lines 5-16 of Algorithm 2, *eTainter* addresses this challenge by recursively checking for possible taint flows that could reach these storage slots.

eTainter first marks these storage slots as sinks (*orders[game].length* in the example), and then performs taint analysis to check if tainted data is written to these storage slots. If so, then the storage slots under analysis are confirmed as taint sources. However, in some cases, *eTainter* cannot locate the `SSTORE` instructions writing to these specific storage slots to mark them as sinks (some storage addresses used in SSTORE instructions are resolved through constant propagation during taint analysis as we will discuss in the following section 5.5.1). *eTainter* gets around this limitation by extracting all paths leading to all `SSTORE` instructions in the contract's bytecode. Later, during taint analysis, the storage addresses are resolved, and *eTainter* can check the flow of tainted data to only the storage slots under analysis.

Handling paths leading to all `SSTORE` instructions would be rather expensive. However, extracting paths leading to `SSTORE` based on slices containing taint sources would reduce the number of the extracted paths. Further, when the taint analyzer finds a vulnerable path, it will stop the analysis of the remaining paths. We found the performance to be quite reasonable in our experiments.

Dealing with Hashed Addresses

Hashed addresses used to reference individual items within arrays and mappings pose two main challenges in propagating taints through the contract storage: (1) difficulty to know the parent array or mapping of an item from its hashed address; and (2) dealing with the cases when the calculation of a hash address depends on user inputs.

To explain the first challenge, in our running example, the variable "fee" is stored in the storage at address 0 and the array "orders[]" is stored starting, say, at address 2. The first slot of the array (address 2 in this example) is used to store the length of the array. However, the addresses of the array's elements are calculated using the EVM instruction `SHA3`, which computes the *Keccak-256* hash of the array slot storing array length (slot 2). Then the element's index is added to the calculated hash to obtain the element's address. For instance, the array element $orders[][1]$ will be stored in the address calculated as $SHA3(orders[].length) + 1$. The loop at line 21 is unbounded by the dynamic array "*orders[].length*" that grows over time in the public function *buyTicket* (line 10). When the function

buyTicket is called, an item will be added to the array and stored in a new storage slot, for example at address 'X', and no tainted data will propagate to the storage slot storing "*orders[].length*", as this slot will be incremented by a *constant* value 1. Therefore, it is not straightforward for *eTainter* to decide that this address is part of the array "*orders[]*", and that tainting the slot at 'X' will also taint the slot storing "*orders[].length*".

To address this challenge, *eTainter* propagates the taint to the slot that stores the length of an array/mapping when one of its slots is tainted. To do this, *eTainter* keeps track of the addresses of the base slots used for calculating storage hash addresses (e.g., the base slot for the hash address calculated through "$SHA3(1)+2$" is 1) when propagating taints. Thus, when *eTainter* finds that the storage slot at address $SHA3(1)+2$ is being tainted, it will also mark the storage slots of the array with the base address "1" as tainted.

As for the second challenge, when propagating taints through storage arrays and mappings, not all the addresses used by storage instructions (`SLOAD` and `SSTORE`) are constants (stated in the bytecode). To handle these cases, *eTainter* resolves the address used in the `SLOAD` or `SSTORE` instruction by propagating constants through the other EVM instructions that derive the address. When *eTainter* cannot resolve the address due to a dependency on user data inputs, which makes precise modeling infeasible, it over-approximates and considers the whole array or mapping as tainted. In our analysis, we preferred to over-approximate for two reasons. First, adding elements to an array results in increasing the array's size, and most of the *unbounded loops* occur due to being bounded by the array sizes of dynamic arrays. Hence, ignoring these cases when addresses are unresolvable will result in several false negatives. Second, the "*DoS with failed call*" vulnerabilities can result from the attacker tainting a single element of the array/mapping that stores the target addresses of external calls performed within loops. Hence, we over-approximate to detect these cases when we are not able to resolve the address for an individual element of an array/mapping.

In EVM, the base addresses of arrays and mappings are always constant, and as mentioned above, *eTainter* keeps track of the arrays and mappings base addresses. This enables *eTainter* to identify the arrays or mappings to which an item belongs even when the item index/key is not a constant.

Handling Memory Taints

eTainter propagates taints through the contract's memory since few instructions use memory as input/output buffer. The challenge faced for modeling memory taints propagation statically in EVM bytecode is that not all offsets used in memory instructions are constant values. In our analysis, *eTainter* implements memory modeling that favors precision. *eTainter* resolves offsets by propagating constants through code instructions that derive memory offsets for memory-based instructions, and it handles memory locations accessed by instructions with unresolved offsets as untainted. This modeling might be incomplete, but it covers the needs of our analysis.

Previous work [154] showed that 80% of offsets in memory writing instructions (`MSTORE`) and 85% in memory reading istructions (`MLOAD`) are statically resolved. Further, most of the unresolved offsets are in instructions that use memory to call other contracts (`CALL` and `DELEGATECALL`), log events (`LOG1`), and halt (`RETURN`) or revert (`REVERT`) contract execution [98]. *eTainter* performs intra-contract analysis and is hence not affected by calling other contracts, and the other instructions either end (`RETURN` and `REVERT`) or do not influence contract execution (`LOG1`).

5.5.2 Checking for Protective Patterns

To enhance the precision of *eTainter* in reporting true vulnerabilities, the taint analysis excludes vulnerable paths that implement the following protective patterns.

Use of Access Control

One of the common practices in smart contracts is the use of function modifiers (Solidity's special construct) to restrict the call of public functions to only the contract's owner or specific addresses. A function modifier checks a condition before the execution of the function. In addition to the use of modifiers, developers could implement checks on the function's caller within the function code itself, such as "*require(msg.sender == owner)*" that halt the execution of the function, and reverts changes when the function is called by any address except the owner. *eTainter* excludes such vulnerabilities that can be exploited only by the contract's owner or authorized users.

Resumable Loops

To protect against DoS attacks when executing unbounded loops, developers may write loops that are split across multiple transactions. The common practice is to check the available gas in each iteration, store the last successful point of the loop in a storage variable, and resume the loop from this point in the next run. *eTainter* excludes vulnerable paths leading to loops implementing these patterns by looking for the code patterns that check available gas either in the loop header or in the loop body. Gas checks might have other uses as well; however, gas checks within loops are usually implemented to prevent DoS attacks, and hence *eTainter* does not report these loops.

5.5.3 Deriving Bytecode's Loops

To define sinks for the unbounded loops, *eTainter* needs to derive loops from the bytecode. However, smart contracts' bytecode often contains several benign loops generated by the compiler that iterate through dynamic arrays, which do not correspond to any iterative pattern in the source code. These loops are used for cases such as handling operations over data items of type strings and bytes (in EVM, data items of type strings and bytes are represented as dynamic arrays). Identifying these loops as vulnerable by *eTainter* would result in several false-positives. Hence, *eTainter* uses static-analysis-based filters (during the definition of the sinks) to filter out these loops. The filters rely mainly on the observation that these loops have a simple structure (exactly two basic blocks; a header and body), and have unique instruction patterns. These patterns are unlikely to overlap with user-defined loops because of the simple functionality of these loops (e.g., initialize a string data item).

For example, in the code: **function** *set(string _n)* {*n = _n;*}
the compiler forms a loop to assign the string received as a parameter (*_n*) to the storage variable (*n*). The simplified body of this loop (does not involve stack instructions) matches the pattern (`MLOAD SSTORE ADD ADD JUMPI`), and is preceded by a code pattern that pushes the length of the string (n) on the stack. Therefore, *eTainter* filters this loop out.

5.6 Evaluation

Our evaluation aims to answer the following four research questions (RQs):

RQ1. What is the effectiveness of *eTainter* in detecting gas-related vulnerabilities compared with MadMax?

RQ2. What is the precision of *eTainter* in detecting gas-related vulnerabilities and code smells?

RQ3. What is the performance of *eTainter* in terms of its analysis time and memory consumption?

RQ4. What is the prevalence of the gas-related vulnerabilities and code smells addressed by *eTainter* in real-world Ethereum contracts?

5.6.1 Experimental Setup

Datasets

To conduct our experiments, we used four datasets as shown in Table 5.3. The first is an annotated dataset consisting of 28 unique smart contracts, and we use it to compare *eTainter* with MadMax [76]. Eight of these contracts had been previously used to evaluate the precision of MadMax [76], which we obtained from the authors - we call these *MadMax contracts*. MadMax had originally been evaluated on 13 contracts, but we excluded five of the contracts that are vulnerable to loop overflows. This is because loop overflows due to casting are statically detected by the current versions of Solidity compilers [77], i.e., they result in compilation errors. The other 20 contracts are from those deployed on the Ethereum blockchain, selected randomly from among the contracts flagged by MadMax as vulnerable. We call these the *random contracts*.

We have manually inspected all the 28 contracts and annotated all the vulnerabilities that belong to the two classes supported by MadMax: (1) unbounded loops, and (2) DoS with failed call. The inspection process is done by two researchers to avoid bias, one of whom is this paper's co-author. Each researcher separately annotated vulnerabilities, and finally, only the vulnerabilities agreed on by both the researchers are annotated in the contracts. We call this the *"Annotated dataset"*.

The second and the third datasets are obtained by downloading a snapshot of the Ethereum blockchain, and extracting the contracts from it using Ethereum Etl [75], an open-source tool. We have extracted $856,121$ contracts from the

Table 5.3: Datasets used in our evaluation.

Dataset	Number of unique contracts
Annotated dataset	28
Ethereum dataset	60,612
Solidity dataset	15,453
Popular-contracts dataset	3,000

snapshot taken on Jan 30, 2021. We removed the duplicates through running the *md5sum* checksum (as done by Duriex et al. [47]) on the contracts' binary code, we ended up with 60,612 unique contracts. We refer to this as the *"Ethereum dataset"*. Of these contracts, 15,454 contracts have their source code available on Etherscan, and we refer to this as the *"Solidity dataset"*.

Finally, we extracted 3,000 contracts deployed on Ethereum blockchain that have the largest number of transactions. These have 89,000 transactions on average, while the average transactions overall is only 129. We call this the *"Popular-contracts dataset"*.

Methods and Metrics

We run all experiments on ten Intel Xeon 2.5 GHz machines, allocating one core and 48 GB of RAM for each run on each machine. In our experiments, we set a timeout value of 5 minutes per smart contract.

To answer **RQ1**, *eTainter* is compared with the MadMax tool in terms of its effectiveness in detecting gas-based vulnerabilities on the Annotated dataset. For a fair comparison, we consider only the vulnerability classes supported by MadMax. We inspect the reported vulnerabilities by MadMax and *eTainter* to determine if these reported vulnerabilities are true-positives or false-positives, based on the annotations of the vulnerabilities in the contracts. We use the inspection results to estimate the precision, recall, and the F1 score (harmonic mean of the precision and recall) for both tools. Further, to remove any bias due to the choice of contracts, we estimate the precision, recall, and the F1 score separately based on the inspection results of (1) MadMax contracts and (2) the random contracts. In our comparison, we use the current version of MadMax [73], along with MadMax's online deployment [163] as it prints analysis logs in a readable format.

To answer **RQ2**, we ran *eTainter* on all the contracts in the Solidity dataset. For the contracts for which *eTainter* reported a vulnerability or code smell, we randomly selected a set of 15 samples from each vulnerability and code smell class. However, some contracts may have more than one vulnerability or code smell of different classes; we thus ended up with more than 15 cases for some classes. In total, we obtained 34 contracts containing 75 reported vulnerabilities and code smells as follows: 16 "unbounded loops", 15 "DoS with failed call", and 15 "unhandled failed calls" vulnerabilities, in addition to 29 "hardcoded gas calls" code smells. We use this set of samples to estimate the precision of *eTainter* by manually inspecting the samples to determine if they are indeed true positives. The precision is the fraction of the true positives out of all reported cases. To avoid bias, the inspection of the selected cases has been done by two researchers independently. Both the researchers work on finding bugs in smart contracts (one is not an author of the paper); and hence have sufficient knowledge in this field. Only cases agreed on by the two inspectors as vulnerabilities and code smells get counted as true positives for estimating the precision.

To answer **RQ3**, we use *eTainter* to analyze each contract in the Ethereum dataset, and calculate the average time for the contracts that were analyzed successfully. If the analysis exceeded the timeout value, we terminate it, and consider *eTainter* as unable to analyze the contract. We also measure its memory usage.

To answer **RQ4**, we ran *eTainter* on the Ethereum dataset to determine how prevalent are the vulnerability classes "unbounded loops," "DoS with failed call," and "unhandled external calls" in real-world contracts. Further, we study how prevalent are "hardcoded gas calls" code smells in real-world contracts. For each vulnerability and code smell class, we count the contracts that contain at least one instance of the class. Similarly, we run *eTainter* on the Popular-contracts dataset to determine how prevalent are "unbounded loops," "DoS with failed call," and "unhandled external calls" vulnerabilities in the widely-used contracts on the Ethereum blockchain, and how prevalent are "hardcoded gas calls" code smells in these contracts.

Table 5.4: Summary of comparison results of *eTainter* with MadMax for the annotated dataset. #N= Reported vulnerabilities. TP = True Positive. FP = False Positive.

	MadMax Contracts							**Random Contracts**						
Vulnerability	**Annotated**	**MadMax**			***eTainter***			**Annotated**	**MadMax**			***eTainter***		
		#N	TPs	FPs	#N	TPs	FPs		#N	TPs	FPs	#N	TPs	FPs
Unbounded loops	4	7	3	4	5	4	1	33	36	23	13	35	31	4
DoS with failed call	1	1	1	0	1	1	0	12	13	10	3	11	11	0
Precision	4/8 (50.0%)				5/6 (83.3%)			33/49 (67.3%)				42/46 (91.3%)		
Recall	4/5 (80%)				5/5 (100%)			33/45 (73.3%)				42/45 (93.3%)		
F1 score	61.5%				90.9%			70.2%				92.3%		

Table 5.5: Overall comparison results of *eTainter* with MadMax for the annotated dataset.

Overall		
	MadMax	***eTainter***
Precision	**37/57 (64.9%)**	**47/52 (90.4%)**
Recall	**37/50 (74%)**	**47/50 (94%)**
F1 score	**69.2%**	**92.2%**

5.6.2 Results

RQ1 (Comparison with MadMax)

Table 5.4 shows a summary of the comparison of *eTainter* with MadMax on the Annotated dataset. The first part shows the comparison results using the MadMax contracts, while the second part shows the comparison results using the random contracts. In each part, the column (Annotated) shows the number of vulnerabilities in the contracts for each class. The other columns show the vulnerabilities reported by both tools (#N), the true-positives (TPs), and the false-positives (FPs). The table shows the precision, recall, and F1 score for each tool. Further, Table 5.5 shows the overall precision, recall, and F1 score for each tool on the annotated dataset.

eTainter *reported 36 of the 37 true vulnerabilities reported by MadMax, and 11 additional true vulnerabilities.* In all, *eTainter* reported *47 out of the 50 vul-*

nerabilities annotated in the 28 contracts (we later discuss the reason for the three false-negatives). Thus, *eTainter* has an overall **precision** of **90.4%**, a **recall** of **94%**, and an **F1 score** of **92.2%**. In comparison, MadMax had a precision of 64.9% and a recall of 74%, which leads to an F1 score of 69.2%.

Examining the results by vulnerability class, for *unbounded loops*, MadMax exhibits low precision (60.5%) and recall (70.2%). In contrast, *eTainter* exhibits high precision (87.5 %) and recall (94.6%) for this class. For *DoS with failed call*, *eTainter* exhibits perfect precision (100%) and a high recall (92.3%), while the precision and recall of MadMax for this class are 78.5% and 84.6%, respectively.

Table 5.4 shows the results of the tools for both MadMax contracts and the random contracts separately to study whether the choice of the random contracts biases the results. Madmax performed slightly better on the random contracts than on the MadMax contracts (F1 score of 70.2% compared to 61.5%). However, *eTainter* performed similarly on both sets (F1 score of 90.9% on the Madmax contracts and 92.3% on the random contracts). Thus, we believe that the choice of the random contracts did not introduce bias.

Note that in our comparison with MadMax on the MadMax contracts, we could not reproduce the results reported in MadMax's paper using the current version of MadMax available on GitHub (commit#6e9a6e99c6) [73]. Specifically, the number of reported false-positives by the current version is higher (MadMax's paper reported only false-positives). When we contacted MadMax's authors about this, they clarified that the main difference is that the current version of MadMax considers that DoS attacks by owners of a smart contract do not count as a vulnerability. However, we do not believe that this was the reason for the difference in MadMax's results, as we did not consider these as true bugs in our annotation of the contracts (and neither does *eTainter*).

To understand the false negative cases of Madmax for all annotated contracts, we find that MadMax did not detect five unbounded loops (reported by *eTainter*) controlled by dynamic items defined within complex structures. Further, all unbounded loops that are controlled by data values stored in memory that reference storage data items are not detected by MadMax - these resulted in three undetected cases. Finally, MadMax's inference rules do not detect DoS with failed call if the Ether transfer is performed by internal functions, called inside the loop, rather than

directly within the loop body. An example is shown in Figure 5.3. The loop at line 8 calls the internal function *send()* at line 10, to send Ether. However, this is not detected by MadMax. In contrast, *eTainter* tracks data flow throughout the contract, and can hence detect these vulnerabilities.

```
1 // Safely sends the ETH by the passed parameters
2 function send(address _receiver, uint _amount) internal {
3    if (_amount > 0 && _receiver != address(0)) {
4        _receiver.transfer(_amount);
5    }
6  }
7 function placeBets() internal {
8    for (uint i=currentIndex; i < bets.length; i++) {
9         Bet memory bet = bets[i];
10        send(bet.player, payout);
11   }
12 }
```

Figure 5.3: A vulnerability found by *eTainter* but not MadMax.

We also find that many false positive cases reported by MadMax were bounded loops that exist only in the bytecode. Further, several other false-positives were due to loops iterating through static arrays that do not grow over time. Figure 5.4 shows an example, where the loop at line 2 is reported by MadMax as an *unbounded loop*, as it iterates through the storage array *owners*. – MadMax considers a loop using the size of a storage array (referenced by any storage write instruction) in its exit condition as an unbounded loop. This array is populated by calling the private function *addOwner_*, which is called only within the constructor,[1] and is hence not vulnerable. In contrast, by finding no dataflow leading to the storage array *owners*, *eTainter* concludes that the *owners* array does not grow over time, and hence does not report it. Thus, tracking data flow results in a more precise approach to detect gas-based vulnerabilities.

As mentioned, Table 5.4 shows that *eTainter* has two false negative cases of unbounded loops and one negative case of DoS with failed call. By analyzing these cases we found that the loop in one case was not defined as a sink by *eTainter* as its form overlaps with a filter that *eTainter* uses to exclude benign loops used to manipulate strings (discussed in 5.5.3). Therefore, the vulnerability is not detected

[1] A contract constructor is executed only once during the contract creation, and its code is not part of the runtime bytecode deployed on the blockchain

```
1  function removeOwner(address owner) external onlyOwner {
2     for (uint i = 0; i < owners.length; i++)
3         { // some code//}
4   }
5  function addOwner_(address owner) private {
6     if (!isOwner[owner]) {
7         isOwner[owner] = true;
8         owners.push(owner);
9     }
10  }
```

Figure 5.4: False positive case reported by MadMax.

by *eTainter*. While we can potentially detect this by enhancing the sinks defining module, it may increase the false positive rate of *eTainter*. Therefore, we did not implement it. The second case was not detected due to an internal error in the code of the tool *rattle* used by *eTainter* to build the SSA form. This was also the cause behind the undetected case of DoS with failed call.

Further, *eTainter* reported five false positive cases as "Unbounded loops". After looking into the code of these contracts, we found that two cases were reported in a contract, in which an array size was used as a bound for the two loops. However, the code limits the number of elements that can be added to the array, thus preventing it from growing in an unbounded fashion. This occurs as *eTainter* does not take into account any sanity checks for sizes of the arrays used in loops, as it is not straightforward to decide when a check is valid. However, blindly excluding all such checks by *eTainter* can result in false-negatives.

Another false positive case is the code pattern "*lastIndex = airdrops.length++;*". The reason for this case is that this code pattern increases the length of the array *airdrops* by 1 and forms a loop in the bytecode similar to "*while (i < airdrops.length) {i++; some code}*", and the value of "i" is set to "*airdrops.length+1*", as shown in the simplified SSA form of the bytecode in Figure 5.5. Although the size of the array, "airdrops", is unbounded, the value of "i" makes the loop condition evaluate to false; it only evaluates to true in the rare case of overflow of "*airdrops.length+1*". Static analysis approaches such as *eTainter* cannot typically handle such dynamic constraints imposed at runtime, and they will hence result in false positives.

The remaining two false positive cases of *eTainter* arise due to how the EVM deals with strings. As mentioned in Section 5.4, *eTainter* filters out loops intro-

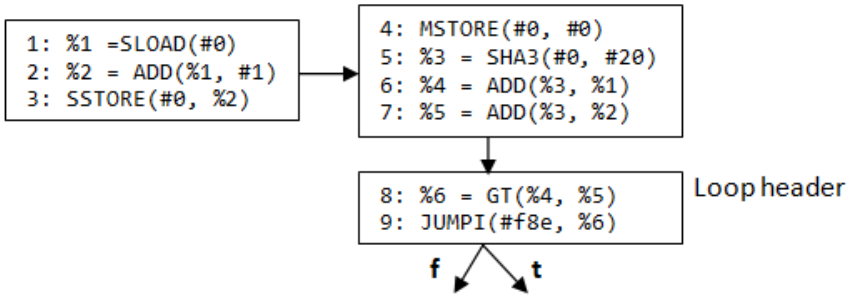

Figure 5.5: Another false positive case reported by *eTainter*.

duced in the bytecode due to string manipulation in the Solidity source code. However, in some cases, the filtering fails, and the loops are incorrectly reported as they use the dynamic sizes of the string arrays as loop bounds.

To sum up, the results of evaluating the effectiveness of *eTainter* in detecting gas-related vulnerabilities (in terms of recall, precision, and F1 score) support our assumption that tracking taints in the contract code is sufficient for finding gas-related vulnerabilities. *eTainter* has both high recall (94%) and precision (90%), which outperformed the prior work, relying on pre-defined patterns. Although *eTainter* has few cases of false-negatives and false-positives, as discussed above, we noticed that most of these cases are due to implementation issues (e.g., internal errors, imprecise identification of the sinks) rather than due to the inapplicability of taints tracking for these cases.

Answer to RQ1: *eTainter* finds *higher* number of true vulnerabilities than MaxMax, and has *fewer* false-positives. Thus, it has both higher recall and precision than MadMax.

RQ2 (Precision of *eTainter*)

As explained in Section 5.6.1, we manually inspected a set of 75 samples reported by *eTainter* in 34 contracts to determine *eTainter*'s precision. Table 5.6 shows a summary of the inspection results.

We find that the overall precision of *eTainter* on the Ethereum dataset is 90.6% across all the contracts. Further, we find that the precision is high (above 80%) for

Table 5.6: Precision of *eTainter* by vulnerability & code smell class.

Class	Inspected	True positives
Unbounded loops	16	14
Loops with external calls	15	14
Unhandled external call	15	12
Hardcoded gas calls	29	28
Precision = 0.906	75	68

all individual classes as well (unbounded loops: 87.5%; DoS with failed call: 93%; unhandled external calls: 80%; and hardcoded gas calls: 96.5%).

An example of the reported vulnerabilities by *eTainter* is the unhandled external call vulnerability shown in Figure 5.6. In this contract, the function *withdraw()* is used by the owner of this contract to transfer the funds of the contract (contract balance) to the address provided by the caller (*'to'* parameter) at line 2. The code does not check the return value of the *send()* function for failures. Thus, if the *send()* failed for any reason, the owner will lose all their funds in the contract, as the execution does not abort after the call failure, and the transferred Ether amount gets deducted from the contract balance. The code also has a hardcoded gas call code smell reported by *eTainter*, in which *send()* limits the gas of the call to 2300 gas units. Thus, even if the contract owner provided high gas for executing the *withdraw()* function, only 2300 gas units gets dedicated to execute *send()*, and in the case of *send()* failure, *withdraw* function resumes execution.

```
1 function withdraw(address to) onlyOwner returns (bool result){
2    to.send(this.balance);
3    return true;
4 }
```

Figure 5.6: Unhandled external call case reported by *eTainter*.

The lowest precision for *eTainter* was for the category "Unhandled external calls," where three of the 15 reported cases were false-positives. The inspection results of these three cases show that in two cases, the code checks that the remaining available gas of the transaction is greater than a specific threshold value before making the external call; to make sure there is enough gas for the call. Figure 5.7

shows the code snippet for one of these cases. In this example, the contract checks that *gasleft()* is greater than or equal to the *MINIMUM_CONSUMER_GAS_LIMIT* constant value (line 3). We consider these two cases as false-positives as the developers implemented countermeasures. However, providing a specific gas amount for the call does not guarantee the successful execution of the call, as the total gas cost of the callee contract is not known and is subject to change. Thus, we did not update *eTainter* to exclude these cases when such countermeasures are found since this may lead to missing true vulnerabilities.

```
1 withdrawableTokens = withdrawableTokens.add(_payment);
2 delete commitments[_requestId];
3 require(gasleft() >= MINIMUM_CONSUMER_GAS_LIMIT, "Must provide consumer
      enough gas");
4 return _callbackAddress.call(_callbackFunctionId, _requestId, _data);
```

Figure 5.7: False positive case of unhandled external call

In the third case, the code uses the external call to submit transactions to the blockchain network, and the contract does not make any changes to its state. Thus, any failure of the external call will neither cause inconsistent changes to the contract state nor can it be exploited by attackers.

Answer to RQ2: *eTainter* has an overall precision of 90.6% in detecting gas-based vulnerabilities and code smells.

RQ3 (Performance of *eTainter*)

eTainter timed out in 12% of the contracts in the Ethereum dataset with a timeout value of 5 minutes. We find that the timeouts in 6,210 (10.3%) out of 7,279 contracts (12%) happen during the CFG generation by *teEther* [96], and conversion to SSA form by *rattle* [74]. We experimented with larger timeout values (15 minutes and 90 minutes) for a set of 1000 contracts, but this did not substantially increase the number of the contracts analyzed successfully (the number of such contracts increased by less than 1% in both cases). We ran MadMax on the Ethereum dataset, and we find that MadMax timed out in only 1.4% of the contracts; thus MadMax scales better than *eTainter*.

For the remaining contracts that were analyzed successfully, the average time of analysis by eTainter*, including CFG generation and conversion to SSA form, is 8 seconds. This is comparable to the average time taken by MadMax (6s).* The minimum time taken by *eTainter* is 0.1 second, and the maximum time is 300 seconds (5 minutes). The average memory consumption of *eTainter* is 118MB (maximum of 1.77GB). *Finally, 96.97% of the contracts that* eTainter *analyzed successfully had an analysis time of less than 60 seconds.*

Answer to RQ3: *eTainter* has an average analysis time of 8 seconds, and a memory consumption of 118MB per contract.

RQ4 (Prevalence of Gas-Related Vulnerabilities and Code Smells)

Ethereum Dataset: *eTainter* flagged 4020 contracts in the Ethereum dataset as having vulnerabilities belonging to the three classes, namely unbounded loops, DoS with failed call, and unhandled external calls, which constitutes 6.6% of the contracts in the dataset. Further, *eTainter* flagged 15,570 contracts as having the harcoded gas smell, which constitutes 25.69% of the contracts in the dataset.

Table 5.7 shows the percentage of contracts with vulnerabilities and code smells for each class in the Ethereum dataset. *The results indicate that all the addressed vulnerability and code smell classes are highly prevalent in real-world contracts, even after excluding the fraction of estimated false-positives.*

We find that 4.1% of the contracts contain unbounded loops, and 1.2% contain loops with external calls that perform a bulk transfer of Ether to several addresses.

Table 5.7: Percentage of contracts fallged by *eTainter* in the Ethereum and Popular-contracts datasets.

Class	Ethereum	Popular-contracts
Unbounded loops	4.1%	1.8%
DoS with failed call	1.2%	0.8%
Unhandled external call	3.2%	2.1%
Hardcoded gas calls	25.7%	22.4%

These contracts are vulnerable to DoS attacks that can result in blocking the use of the contracts forever. Further, 3.2% of contracts have unhandled external calls vulnerabilities, and these calls can cause unexpected behavior in the contracts logic if the calls failed either due to attackers or accidentally. Finally, we find that 25.7% of the contracts use external calls with hardcoded gas. Therefore, any future changes in the gas cost of EVM opcodes can block a large share of these contracts from, for example, sending Ether if they are not upgradeable.

Popular-Contracts Dataset: Table 5.7 shows the percentage of the contracts with vulnerabilities and code smells for each class in the Popular-contracts dataset. In this dataset, *eTainter* flagged 144 contracts (4.8% of the contracts) as having "unbounded loops", "DoS with failed call", and "unhandled external calls" vulnerabilities. Further, results show that 22.4% of the contracts have "hardcoded gas calls" code smells. The percentage of contracts vulnerable to "Unbounded loops", "DoS with failed call", and "Unhandled external calls" is lower in the Popular-contracts dataset (1.8%, 0.8%, and 2.1%, respectively, compared to 4.1%, 1.2%, and 3.2% in the Ethereum dataset). This result is along expected lines because contracts in the Popular-contracts dataset are less likely to be vulnerable than other contracts [136], as these contracts usually undergo manual audits before deploying. On the other hand, similar to the Ethereum dataset, the percentage of contracts with "hardcoded gas calls" code smell is high in the most widely used contracts. This suggests that even manually audited contracts do not adopt security best practices.

Answer to RQ4: All four classes of gas-related vulnerabilities and code smells are prevalent in both sets of contracts. *eTainter* finds unbounded loops, DoS with failed call, and unhandled external calls in over 6.6% of the contracts in the Ethereum dataset and in 4.8% of the most frequently used contracts. Further, 25% of the Ethereum dataset contracts and 22% of the frequently-used contracts have "hardcoded gas calls" code smells.

5.7 Threats to Validity and Limitations

Threats to Validity

An *External* threat to validity is the limited number of smart contracts (28) used to compare *eTainter* with MadMax. This is due to the time needed to inspect contracts manually (by two researchers) and annotate the vulnerabilities in them, due to the lack of any publicly labelled dataset for the same. We have partially mitigated this threat in two ways. First, we have included all the contracts that were used in MadMax's paper [76] (barring those that no longer compile). Second, the remaining 20 contracts were selected randomly from the Ethereum blockchain.

An *Internal* threat to validity is the potential bias in annotating the vulnerabilities in the 28 smart contracts of the annotated dataset. We have mitigated this threat by having two researchers in the area performing the annotation independently, and including only the vulnerabilities agreed on by both researchers as true vulnerabilities.

Another *External* threat to validity is the limited number of selected cases we have used to estimate the precision of *eTainter* in RQ2. This is because manual inspection of smart contracts is a very time-consuming process. We have mitigated this threat by randomly sampling from the different vulnerability and code smell classes, and selecting a minimum of 15 reported cases for each class.

Limitations

eTainter works on finding gas-based vulnerabilities caused due to dependency on user data. However, there might be unbounded loops that depend on data items (e.g., arrays) growing over time by adding new items of constant values into the array, which would not be detected (we did not find any such cases).

Another limitation is that *eTainter* uses other tools to generate the CFG, and lift the bytecode to the SSA form. These tools caused a timeout in many contracts (10.3%), and hence *eTainter* was not able to analyze these contracts.

5.8 Summary

This chapter proposed *eTainter*, a static analysis approach for finding four classes of gas-related vulnerabilities and code smells in smart contracts, using static taint analysis on the contract's EVM bytecode. We evaluated *eTainter* on a set of 28 annotated contracts as well as on over $60,000$ unique, real-world contracts deployed on Ethereum. The results show that *eTainter* is able to find gas-related vulnerabilities and code smells with a precision of over 90% and an average time of 8 seconds per smart contract. Further, the results show that *eTainter* has higher recall than the prior work, MadMax, on detecting gas-related vulnerabilities and achieves higher precision as well. Finally, gas-related vulnerabilities and code smells are prevalent in real-world smart contracts on Ethereum, including the most frequently used smart contracts.

The following chapter (Chapter 6) presents our second vulnerability detection approach, *AChecker*, for finding access control vulnerabilities in smart contracts.

Chapter 6

Detecting Smart Contract Access Control Vulnerabilities

In Chapter 5, we presented our first approach that addresses the problem of effectively finding vulnerabilities in smart contracts. In this chapter, we discuss our second approach *AChecker*, which targets detecting access control vulnerabilities. We first discuss the challenges of statically finding access control vulnerabilities in smart contracts. We then illustrate through examples our classification of access control vulnerabilities and the limitations of the underlying techniques of the prior work. Next, we discuss our insights to effectively detect access control vulnerabilities and present our proposed techniques based on these insights. Finally, we discuss the evaluation results.

6.1 Introduction

As smart contracts can be called by any user on the blockchain, as discussed in Chapter 1 (Section 1.2.3), contract developers typically implement access control to manage who can call specific functions within the contract or execute critical actions, such as withdraw money or destroy the contract and remove it from the blockchain. Developers implement access control checks in different methods, such as through the use of language-special-constructs, e.g., function modifier, or assertions and conditional statements, e.g., *require(access-control-condition)*.

Failing to implement access control checks properly can result in either (1) weak checks that can be bypassed by unauthorized users, or (2) missing access control for code statements that need to be protected. We call these *access control vulnerabilities*. There have been many instances of real-world attacks due to access control vulnerabilities. For example, in May 2021, an attack targeting a smart contract used by the ValueDeFi platform [127] led to the loss of about 10M. *The attack was due to a mistake in a single line of code that resulted in an access control vulnerability, which enabled the attacker to make themselves an owner of the contract.* Additionally, the attack that targeted Parity Wallet and led to locking out 280M worth of Ether [18] *was due to having an unprotected function in the contract, which enabled the attacker to reset the variables storing addresses of wallet owners and destroy the contract, thereby freezing all the funds in the wallet.*

Static analysis can help identify access control vulnerabilities before the deployment of the contract on the blockchain. Unfortunately, statically analyzing smart contracts for access control vulnerabilities is a challenging problem because: (1) The lack of access control policy specifications make it difficult to precisely identify access control checks. (2) some code patterns that appear to be vulnerabilities may actually be intentional parts of the contract's functionality, and it is difficult to statically distinguish between these code patterns and actual vulnerabilities due to the lack of the contract access control specifications.

In this chapter, we propose an efficient approach, *AChecker* (Access Control Checker), for discovering access control vulnerabilities in smart contracts. Our key insight for identifying access control checks is that they have unique functionality that can be inferred by analyzing data dependencies [81] in the conditions forming the checks. After identifying access control checks, we detect access control vulnerabilities by analyzing data dependencies between the data inputs from the contract users and (a) state variables storing access control data and (b) a set of predefined code statements. The main intuition behind our analysis is that these "critical" instructions can only be manipulated by trusted users, and thus should be protected by access control checks. That is, we formulate the detection of access control vulnerabilities as a taint analysis problem [132], without relying on pre-defined access control patterns or existing transactions.

Furthermore, to avoid reporting false-positive results, our analysis identifies

cases of *potentially intended behaviors*. We assume a behavior is intended when the code implements non-access control constraints that allow manipulating the access control data only under specific conditions. Our intuition behind this heuristic is that having non-access control constraints to guard access control data *under specific conditions* implies that the developer intentionally implemented this code behavior. We employ symbolic execution analysis to synthesize constraints under which manipulation of access control data occurs; we then separate potentially intended behaviors from more certain access control vulnerabilities.

To the best of our knowledge, AChecker *is the first technique that statically reasons about access control checks without requiring either pre-defined code patterns or pre-existing transactions history. Further, the intended behavior is reported by the current analysis tools as vulnerabilities, and* AChecker *is the first to optimize for high-precision by analyzing for intended behavior cases and separating them as potentially intended cases, thereby minimizing false alarms.*

We evaluate *AChecker* on three datasets collected by prior work: the first one, CVE [101], consists of 15 contracts with access control vulnerabilities; the other two datasets, SmartBugs [47] and Popular-contracts [70], are large dataset of real-world Ethereum contracts, with 47,518 and 3,000 contracts, respectively. For our evaluation, we assess the efficiency of *AChecker* by comparing it to that of eight existing tools that target access control vulnerabilities, namely SPCon [101], Ethainter [17], Securify [154], Manticore [106], Maian [113], Mythril [34], Slither [56], and Smartcheck [149].

We summarize our main contributions of this chapter below.

1. Propose a novel static data-flow-analysis-based technique for efficient identification of access control checks, relying on neither pre-defined code patterns nor existing transactions history, unlike other tools.

2. Propose a novel symbolic execution-based approach for reducing false-positives by automatically inferring cases where untrusted users are allowed to manipulate access control data as an intended behavior of the contract.

3. Combine our proposed methods in a consolidated approach, *AChecker*, for detecting access control vulnerabilities. We implement the approach in an

automated tool and make the implementation publicly available [67].

4. Evaluate *AChecker* on three public datasets and show that it outperforms all eight existing approaches used as a baseline, in terms of both precision and recall. Furthermore, it is able to detect a large number of vulnerable contracts in SmartBugs and Popular-contracts datasets with a high precision.

6.2 Motivating Examples

In this section, we use real-world examples to discuss different types of access control vulnerabilities and the challenges of finding them in smart contracts. Moreover, we discuss cases of potentially intended behavior in the contract that are reported as access control vulnerabilities by current analysis tools.

Our definition of access control vulnerabilities is based on prior work [17, 34, 56, 101, 106, 113, 149, 154]. We classify access control vulnerabilities into two main categories, according to their cause. The first group involves vulnerabilities that occur due to vulnerable access control checks that can be bypassed by attackers. We refer to these as "*violated access control checks (VACC)*". The second group consists of the vulnerabilities that arise due to the lack of access control for critical instructions. We refer to these as "*missing access control checks (MACC)*."

6.2.1 Violated Access Control Check (VACC)

The code in Fig. 6.1 is simplified from a real-world contract that implements an Ethereum token for Business Alliance Financial Circle (BAFC) [8]. The contract was disclosed as having an access control vulnerability and is assigned the CVE "CVE-2018-19830". In this contract, the `owner` state variable (defined in line 2) is used to store the address of the owner of this contract, and the `frozenAccount` (line 3) is a state mapping that stores frozen accounts that cannot perform specific actions. The modifier `onlyOwner` (lines 5-11) checks if the caller is the owner, and the modifier `unFrozenAccount` (lines 12-15) checks if the address of the caller is not frozen. The `onlyOwner` modifier is used to protect the functions `freezeAccount`, `switchLiquidity`, and some others (not shown in Fig 6.1). The `unFrozenAccount` modifier is used in several functions, such as `transfer`.

```
1  contract BAFCToken {
2    address owner = msg.sender;
3    mapping (address => bool) public frozenAccount;
4    /****** modifiers ******/
5     modifier onlyOwner {
6        if (owner == msg.sender) {
7            _; }
8        else {
9            InvalidCaller(msg.sender);
10           throw;}
11    }
12    modifier unFrozenAccount{
13       require(!frozenAccount[msg.sender]);
14       _;
15   }
16  /****** Functions *******/
17   function UBSecToken () public {
18      owner = msg.sender;
19      totalSupply = 1.9 * 10 ** 26;
20   }
21   function freezeAccount(address target, bool freeze) onlyOwner public{
22       frozenAccount[target]=freeze;
23     }
24   function switchLiquidity(bool _transferable) onlyOwner{
25       transferable=_transferable;
26   }
27   function transfer(address _to, uint _value) unFrozenAccount
         onlyTransferable {
28     if (frozenAccount[_to]) {
29        InvalidAccount(_to, "Frozen receiver account");}
30     else {
31       balances[msg.sender]=balances[msg.sender].sub(_value);
32       balances[_to] = balances[_to].add(_value);
33     }
34   }
```

Figure 6.1: Real-world contract with violated access control checks.

The function `UBSecToken` (line 17) is used to initialize the `owner` variable and some other data items. This function is supposed to be defined as a constructor to get executed when the contract is deployed to initialize the `owner` with the address that creates (deploys) the contract. As a constructor, it would not be part of the code running on the blockchain. However, the developer made a mistake and named the constructor `UBSecToken` instead of `BAFCToken`; hence, the misspelled constructor leads to the `UBSecToken` function to be callable by anyone. This mistake enables anyone to make themselves an owner of this contract, and perform actions such as switching the status of the token liquidity using the

`switchLiquidity` function (line 24). Furthermore, the user who hijacks the ownership of this contract can freeze/unfreeze any address having tokens managed by the contract, thereby violating the access control check `unFrozenAccount` and becoming able to control who can perform some actions, such as transferring or receiving tokens using function `transfer` (line 27).

We found that Ethainter [17] does not report vulnerabilities in this contract because Ethainter's inference rules do not recognize the access control check implemented by the `onlyOwner` modifier. The code of `onlyOwner` was compiled to a bytecode pattern that does not match the style expected by Ethainter's inference rules due to an extra jump block in the bytecode. Although this case can be fixed by adding more rules, it is difficult to enumerate all such scenarios generated by the compiler, and add inference rules for them.

Similarly, this contract is out of the scope of SPCon [101] as there are only limited transactions on this contract's functions, and as discussed earlier, SPCon mines access control rules from transactions. Thus, SPCon was not able to mine the security policy of this contract to be able to detect the vulnerability. In addition, for scalability, SPCon only considers two transactions executed in sequence to trigger vulnerabilities; thus, it cannot detect access control checks attacked through more than two transactions, such as `unFrozenAccount` getting violated when `onlyOwner` gets violated.

Finally, to the best of our knowledge, no approach finds the mapping between the violated access control checks and the affected functions in the contract; hence, existing tools do not find the contract function(s) that becomes accessible to attackers when an access control check is violated and leave that for developers to check it manually, which is difficult and time-consuming for complex contracts. For example, in the contract in this example, none of the existing tools report that the functions `freezeAccount`, `switchLiquidity`, `transfer`, etc., are accessible by attackers because `onlyOwner` is violated in function `UBSecToken`.

6.2.2 Missing Access Control Check (MACC)

As mentioned earlier, the contract code may contain critical instructions that have to be protected, so they do not get executed or manipulated by attackers [138–140].

For example, in the code excerpt shown in Figure 6.2, taken from a real-world contract [51], the instruction `selfdestruct(msg.sender)` at line 2 will remove the contract from the blockchain and send whatever amount is in the contract balance to the address provided as a parameter to the instruction (`msg.sender`). Therefore, only authorized users, e.g., owner, should be allowed to invoke the function `removeContract`. Further, the parameter of the instruction `selfdestruct` should not be set by unauthorized users.

```
1 function removeContract() public {
2  selfdestruct(msg.sender);//remove contract from network
3 }
```

Figure 6.2: Missing access control check example.

```
1  modifier onlyOwner() {
2   require(owner()==_msgSender(),"Caller is not owner");_;
3   }
4  function owner() public view returns (address) {
5    return _owner;
6   }
7 //@dev Destroy contract and reclaim leftover funds
8  function kill() external onlyOwner {
9    selfdestruct(payable(_msgSender()));
10  }
```

Figure 6.3: Protected `selfdestruct` reported as a vulnerability by prior work.

Several existing tools cannot detect the two vulnerabilities in this example. For instance, the function `removeContract` is called only once to destroy the contract, and hence SPCon cannot detect these vulnerabilities because transactions to the function `removeContract` (needed for SPCon to mine security roles) will only be available on the blockchain when the attack is done and the contract is destroyed. On the other hand, approaches relying on pre-defined patterns report high false-positives when analyzing for these vulnerabilities. For example, in the code in Figure 6.3, taken from an NFT contract called "Sugoi NFT NYC" [145], Ethainter flags the function `kill`, having `selfdestruct` (line 9), as vulnerable even though it is protected by a secure access control modifier `onlyOwner`. We found that Ethainter fails to recognize the modifier `onlyOwner` protecting the

function `kill`. This is because `onlyOwner` calls the `owner()` function to obtain the address of the owner, and Ethainter's inference rules cannot match patterns across functions. Ethainter looks for patterns of access checks within connected code blocks; however, calling another function from within the access check results in jumping into other blocks to execute the function, which makes it difficult for Ethainter to identify the access check.

6.2.3 Potentially Intended Behaviors

In some cases, the contract code may be intentionally implemented to allow any user to update access control state variables under specific conditions. Analysis tools should distinguish these cases from exploitable true vulnerabilities to maximize benefit of the analysis tools. We refer to these cases as *potentially intended behaviors*. For example, in the code excerpt in Figure 6.4, extracted from a real-world contract, the function `changeNameSymbol` has an access control check at line 4, so that only the contract's owner, whose address is stored in the `owner` state variable, can call it. On the other hand, the contract has the function `changeOwner`, which enables anyone to buy the contract's ownership for 1000 Wei.

We find that almost all the existing tools looking for untrusted writes of access control data will report the `changeOwner` function as a vulnerability since anyone can change the `owner` state variable. However, this behavior is intentional: the contract owner intentionally allowed anyone to buy the ownership. When the paid amount gets transferred to the current owner, the `owner` is set to the buyer's address.

```
1  address public owner;
2  uint256 constant howMuchToBecomeOwner= 1000 ether;
3  function changeNameSymbol(string _n, string _s)external{
4     if (msg.sender==owner){
5       /* omitted code */
6   }
7  function changeOwner(address _newowner) payable external{
8     if (msg.value>=howMuchToBecomeOwner){
9        owner.transfer(msg.value);
10       owner=_newowner;}
11  }
```

Figure 6.4: Intended behavior example.

6.3 *AChecker* Approach

6.3.1 Overview of *AChecker*

Figure 6.5 shows the workflow of the proposed approach, *AChecker*. The approach consists of three major steps. (1) *AChecker* takes as input the contract bytecode, and performs static data-flow analysis to identify the access control checks that are implemented and used in the contract, as well as the corresponding state variables storing access control data. *AChecker* also identifies critical instructions that should be protected. (2) Using the identified access control state variables and critical instructions (from the previous step) as sinks, *AChecker* explores possible taint paths from user inputs (taint sources) to the sinks. (3) *AChecker* symbolically executes paths of the taint-flows reaching sinks to filter out infeasible paths and separate intended behavior cases from exploitable true vulnerabilities. Finally, *AChecker* flags the found vulnerabilities, and the corresponding functions to the users. We explain the steps in the following sub-sections.

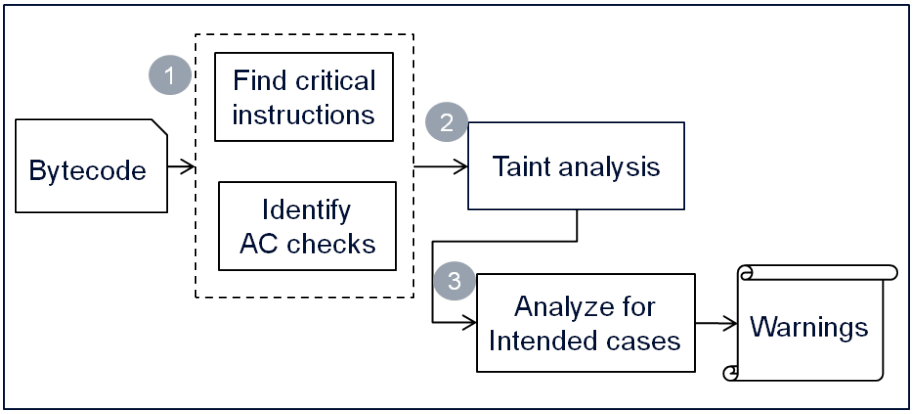

Figure 6.5: *AChecker* workflow.

6.3.2 Identifying Access Control Checks

To find access control vulnerabilities, *AChecker* first identifies access control checks used in the contract, without relying on syntactic patterns of access control checks. Before discussing our approach for identifying access control checks, we formally

define the following based on the functionality provided by access control checks in smart contracts:

Definition 1 (Access Control Check) *Let C be a condition of a conditional control statement in a smart contract S, and C validates the callers of the contract S that can execute the code protected by the condition C, then the condition C is an access control check.*

Definition 2 (Access Control State Variable) *In an access control check C, if C compares the caller of the contract against value(s) stored in a storage data item D or if C uses the contract caller to load a value from a storage data item D to evaluate C, then D is an access control state variable.*

Definition 3 (Authorized User) *In an access control check C that uses an access control state variable D, any user whose address $\in$ D is an authorized user to execute the contract code protected by the access control check C.*

For example, in the contract code in Figure 6.1, the `switchLiquidity` function has the modifier `onlyOwner`. This modifier checks if the address of the caller (`msg.sender`) is equal to the address stored in the state variable `owner` using statement `if(owner==msg.sender)`. The execution of the function `switchLiquidity` happens only when the `if` condition is true. Thus, in function `switchLiquidity`, the condition `owner==msg.sender` is an access control check, the `owner` is an access control state variable, and the user whose address stored in the state variable `owner` is an authorized user to execute the function `switchLiquidity`.

The definitions mentioned above state that, the unique functionality of all access control checks, regardless of how they are written, is to check the callers of the smart contract against the access control state variables (representing various authorized users). Thus, our intuitive idea to identify access control checks in the contract code is to analyze for conditions that reference the contract caller and either compare it to values loaded from the contract storage (i.e., access control state variables) or use it to load values from the contract storage to evaluate the condition. In smart contracts, a built-in global variable is used to obtain the contract

caller – `msg.sender` in the source code and `CALLER` in the bytecode. Therefore, we can leverage this feature to perform our analysis.

Algorithm 3: Identifying Access Control Checks

```
   Input: C ▷ Conditions
   Output: AC, SV ▷ Access control checks and state variables
1  begin
2      AC ← ∅ ▷ Initial set of access control checks
3      foreach condition c ∈ C do
4          flow ← BackwardAnalysis(c)
5          if CALLER & SLOAD ∈ flow then
6              s ← SlotOf(SLOAD)
7              if s ∉ mappings then
8                  AC ← AC ∪ c; SV ← SV ∪ s
9              else if s ∈ mappings & TypeValue(s) ∈ Boolean then
10                 AC ← AC ∪ c; SV ← SV ∪ s
11             else if s ∈ mappings then
12                 t ← ForwardAnalysis(c,s)
13                 if not t then
14                     AC ← AC ∪ c; SV ← SV ∪ s
15     return AC, SV
```

We conduct our analysis to identify access control checks by statically analyzing data-flows of the conditions in the contract to obtain conditions that have data dependency on the two instructions, `CALLER` and `SLOAD` (storage load), as shown in Algorithm 3. The algorithm takes as input the conditions in the contract code and returns as output, the access control checks implemented in the contract and the corresponding access control state variables. For each condition in the contract, *AChecker* first performs backward data-flow analysis starting at the condition (lines 3-4). By considering all data dependencies, the resulting flows contain all instructions that are involved in evaluating the condition. Then *AChecker* filters conditions having data dependency on `CALLER` and `SLOAD` instructions (line 5). These filtered conditions form an overapproximation of access control checks.

For example, in the code in Figure 6.1, the modifier `onlyOwner` (lines 5-11) is used as access control to protect the function `switchLiquidity`. The simplified bytecode of this function, in static single assignment (SSA) form, is shown in Figure 6.6. The code of the first block (lines 1-6) checks if the caller is the owner

(represents the code of the modifier `onlyOwner`). If so, the execution is directed to the block at line 8 to execute the code of the function `switchLiquidity`. Otherwise, the execution goes to the block at line 7, where `REVERT` instruction halts the execution, and reverts changes made to the contract state.

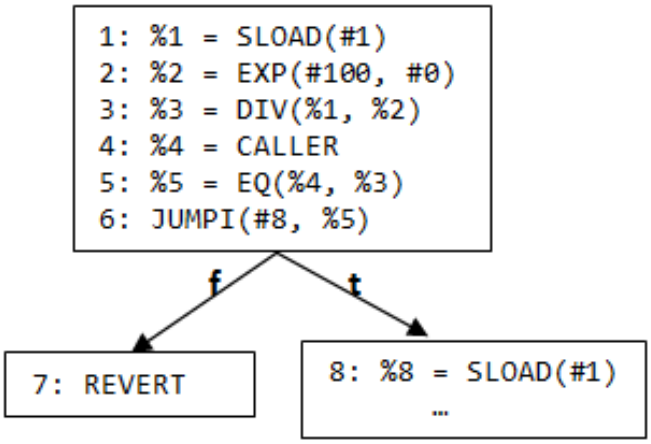

Figure 6.6: Bytecode of the function switchLiquidity in Figure 6.1.

To identify the access control check performed in this function, *AChecker* performs backward data-flow analysis starting at the condition of the `JUMPI` instruction at line 6. *AChecker* will find the data-flow: `%1=SLOAD(#1), %2=EXP(#100,#0), %3=DIV(%1,%2), %4=CALLER, %5=EQ(%4,%3)`. The instructions `%1=SLOAD(#1)` and `%4=CALLER` are part of this data-flow to evaluate the condition of the `JUMPI` instruction. Thus, the condition of the `JUMPI` at line 6 is identified as an access control check, and the storage location #1 (read by instruction `%1=SLOAD(#1)`) is identified as an access control state variable (that should not be written by the contract users).

A challenge faced in our analysis is that the conditions obtained by the backward data-flow analysis are (by design) an over-approximation of the checks (conditions) in the contract code that have data dependency on `CALLER` and `SLOAD` instructions. Thus, the filtered checks may contain non-access control checks that behave similar to access control checks (i.e., reference the contract's caller, and load a value from storage). These cases result from conditions in which the

contract caller is used as a key to access values in storage mappings, such as $require(balances[msg.sender] >= amount)$.

To address this challenge, *AChecker* excludes these conditions from the set of conditions identified as access control checks. The core idea is to use data-flow analysis to differentiate between the mappings representing access control state variables and other mappings. The idea is based on two observations. First, the values of mappings used as access control state variables are only used for validating callers, not for any other operations, e.g., mathematical operations. Second, mappings representing access control state variables usually store Boolean values. Based on these two observations, *AChecker* performs further analysis of the conditions returned by the backward data-flow analysis in which the contract caller is used to access mappings (lines 6-14 in Algorithm 3).

For each condition returned by the backward data-flow analysis, *AChecker* confirms the condition as an access control check if the corresponding state variable used in the condition is not a mapping (lines 7-8). Otherwise, if the state variable is a mapping and stores Boolean values, then *AChecker* confirms the condition as an access control check (lines 9-10). Although type information is not available at the EVM bytecode level, we found that the data type of the values stored in mappings can be inferred from the bytecode. This is because for mappings storing Boolean values, the compiler uses AND bitmasking with Boolean values, where any Boolean value loaded from the storage gets bitmasked using '0xff' (Boolean holds one byte of the storage location). Finally, if the state variable is a mapping and does not store Boolean values, *AChecker* performs static forward data-flow analysis starting at the `SLOAD` instruction to check if the mapping referenced in the condition is used by any other operation following the condition. If no such data-flow is found, the condition gets confirmed as an access control check (lines 11-14). The analysis stops when reaching a `STOP` or `RETURN` instruction.

After identifying access control checks and the corresponding state variables, *AChecker* searches for critical instructions in the contract code and proceeds to the next step.

6.3.3 Violated/Missing Access Control Checks Detection

In the second step, *AChecker* performs static taint analysis to check if any of the access control state variables and critical instructions are manipulated by attackers. The performed taint analysis is both inter-procedural and context-sensitive, and tracks taints through the contract storage.

The analysis uses user inputs as sources of taint, and uses critical instructions along with access control state variables as sinks. Table 6.1 shows the sinks used by *AChecker* for each vulnerability. Each `SSTORE` instruction that writes to any of the identified access control state variables is used as a sink to detect violated access control checks. Further, `SELFDESTRUCT` instructions, and the address parameters of `SELFDESTRUCT` and `DELEGATECALL` instructions are used as sinks to detect missing access control checks, as the target addresses of `SELFDESTRUCT` and `DELEGATECALL` should not be manipulated by unauthorized users. If the attacker is able to control the address parameter of `SELFDESTRUCT`, they will receive the funds of the contract when the `SELFDESTRUCT` is executed, as discussed in Section 6.2.2. Similarly, setting the address of the `DELEGATECALL` will allow attackers to execute their own code in the context of the contract, thereby manipulating the contract state and changing its behavior.

Table 6.1: EVM instructions defined as sinks by *AChecker*. The argument in **BOLD** is the sink; otherwise, the whole instruction is a sink.

Vulnerability	EVM Instruction
VAAC	`SSTORE` <key, **value**>, key $\in$ AC state variables
MAAC	`SELFDESTRUCT`
MAAC	`SELFDESTRUCT` <**addr**>
MAAC	`DELEGATECALL` <, **addr**, ...>

A sanitizer is a code check that prevents contract users from either controlling a critical instruction or manipulating an access control state variable. There are two types of *unsanitized* taint flows that are considered by *AChecker*, (1) taint flows to access control state variables are considered as *violated access control checks*, and (2) taint-flows to critical instructions are considered as *missing access control checks*. Finding taint flows to access control state variables and critical instructions results in discovering access control violations as well as missing access control,

where the attacker is able to manipulate the access control state variables or critical instructions. This includes the cases when incorrect access control modifier names that can be bypassed by attackers are used as access control checks.

6.3.4 Potentially Intended Behaviors Filtering

One of the challenges faced when analyzing smart contracts for access control vulnerabilities is the intended behavior cases (Section 6.2.3). Based on our observations, we found that a lack of access control does not always result in a vulnerability, as the developer may implement other non-access control checks. The contract code may *intentionally* allow even unauthorized users of the contract (any Ethereum account) to manipulate access control state variables but only under specific non-access control constraints. These cases are challenging for static analysis approaches to distinguish from vulnerabilities, as there are often no access control policy specifications. Figure 6.4 shows an example where unauthorized users can write to the access control state variable `owner` (line 11) when they pay a minimum of 1000 Wei. Static analysis tools would falsely flag this as a vulnerability.

AChecker addresses this challenge by performing symbolic-based analysis to infer these non-access control checks implemented in these cases, and then flagging these cases as potentially intended behaviors. Our intuition in reasoning about intended behavior cases is that an unprotected statement that manipulates an access control state variable, *but only* under specific non-access control constraints, is likely to be a potentially intended behavior implemented by the developer.

Definition 4 (Potentially Intended Behavior) *In a smart contract, manipulating an access control state variable with no access control check, and under other non-access control constraints, is most likely an intended behavior rather than an access control vulnerability.*

A naive approach to filter potentially intended behaviors would be to flag as potentially intended *any unprotected statement* that updates access control data and is guarded by non-access control checks. However, blindly filtering cases in this way will result in several vulnerabilities falsely reported as intended behaviors. For example, Figure 6.7 shows a vulnerable function `initialize` implemented to initialize the access control state variable `operator` (line 4) while

`initialized` is 'false'. The developer mistakenly did not add a statement to set the `initialized` to 'true' after line 4; which resulted in a vulnerability. However, deciding intended behavior just based on searching for implemented checks will report this vulnerability as an intended behavior.

```
1 bool public initialized = false ;
2 function initialize () {
3    require (! initialized);
4    operator = msg.sender;
5 }
```

Figure 6.7: Example of a vulnerability due to an always True check.

Therefore, to analyze for potentially intended behaviors, *AChecker* uses symbolic execution [92] to infer constraints of manipulating access control state variables, and differentiate potentially intended behaviors from vulnerabilities as below. Our design choice to use symbolic execution is supported by the observation that constraints implemented in intended behaviors are usually few, therefore tractable, and can hence be solved using satisfiability modulo theories (SMT) solvers.

When a taint-flow is found to an access control state variable, *AChecker* executes the taint-flow path symbolically to generate constraints under which the taint flows from the taint source to the access control state variable, then uses an SMT solver to check if the path is feasible. If the path is feasible, *AChecker* uses the SMT solver to find an assignment that makes these constraints unsatisfiable through negating the constraints. If an assignment is found to make the negated constraints satisfiable, then *AChecker* reports this as a potentially intended behavior as the taint-flow is not always feasible – otherwise, *AChecker* reports it as a vulnerability.

For the example in Figure 6.4, *AChecker* finds that `owner` gets tainted at line 10. Then, *AChecker* symbolically executes the taint-flow path to synthesize the constraints under which `owner` gets tainted. In this case, the synthesized path constraints are:
"$msg.value \geq howMuchToBecomeOwner \wedge howMuchToBecomeOwner = 1000$". The path is feasible and the SMT solver will find an assignment for `msg.value` that satisfies the negated constraint "$!msg.value \geq howMuchToBecomeOwner \wedge$

howMuchToBecomeOwner = 1000", as we are interested in proving that the original constraints (without negation) are not always satisfied. *AChecker* will flag this as a potentially intended behavior as it was able to find cases under which manipulating the state variable `owner` is not always feasible. This is because restricting execution of the code updating `owner` state variable by implementing specific constraints is likely intended by the developer to manage `owner` writes.

When a solution is not found to make the constraints unsatisfiable, if the constraints depend on storage variables getting updated in the taint-flow path, *AChecker* calls the SMT solver again using the updated state. We need this to infer the intended behavior cases usually used for initializing storage variables only once and then set a specific storage variable to prevent future initialization. Further, our symbolic analysis abstracts loop analysis where loops get unrolled to at most three iterations to reduce overhead and filter more potentially intended behavior cases. With that said, we observed from our experiments that loops rarely get used in paths that manipulate access control state variables.

6.3.5 Implementation

The implementation of *AChecker* relies on *eTainter* [70], a static taint analyzer for smart contracts, to perform taint analysis, and on *teEther* [96], a symbolic analysis framework for smart contracts, to generate the control flow graph (CFG) and perform symbolic analysis. We built up on the symbolic-execution module of *teEther* to perform our intended analysis, specifically, to unroll loops and negate generated constraints when inferring potential intended behavior.

6.4 Evaluation

We evaluate *AChecker* by answering the following three research questions (RQs):
RQ1. How effective is *AChecker* in detecting access control vulnerabilities compared to the prior work?
RQ2. How precise is *AChecker* in inferring potentially intended behaviors in smart contracts?
RQ3. What is the performance of *AChecker*?
RQ4. How prevalent are access control vulnerabilities in smart contracts?

Table 6.2: Datasets used in our evaluation.

Dataset	Number of Contracts
CVE dataset	15
SmartBugs-wild dataset	47,518
Popular-contracts dataset	3,000

6.4.1 Experimental Setup

Datasets

In our evaluation, we use three datasets, as summarized in Table 6.2. The first is the *CVE dataset* consisting of 15 contracts, where each contract is assigned a CVE (Common Vulnerability and Explosures) for an access control vulnerability. This dataset has been collected by the authors of SPCon [101] to evaluate their tool and compare with the prior work. They originally collected 17 contracts in the SPCon paper, but they reported that one contract is not the vulnerable contract discussed by the CVE-2021-3006 (the disclosing report of this CVE referenced a non-vulnerable contract). Another contract (CVE-2018-17111) has a vulnerability in a modifier that is not used by any function in the contract, so the code of the modifier is unreachable. Therefore, we omitted these two contracts from the dataset.

The second dataset is the *SmartBugs-wild* public dataset which consists of about 47,518 contracts. The SmartBugs-wild dataset has been collected by Durieux et al. [47] to evaluate nine static analysis tools. Finally, the third dataset is the *Popular-contracts dataset* from eTainter [70]. This dataset consists of 3,000 contracts deployed on the Ethereum blockchain, which have the largest number of transactions as of January 2022. Thus, this dataset represents contracts that are highly used in the Ethereum blockchain.

Setup

We run the experiments on five Intel Xeon 2.5 GHz machines; each machine has one core. On each machine, we allocate 48GB of RAM for each run. For all tools, we set a timeout value of 10 minutes (600 seconds) per smart contract.

Method and Metrics

To answer **RQ1**, we compare the effectiveness of *AChecker* by comparing it to eight analysis tools that target detecting all or a subset of access control vulnerabilities. These tools are SPCon [101], Ethainter [17], Securify [154], Mythril [34], Manticore [106], SmartCheck [149], Maian [113], and Slither [56]. We use two different datasets to compare against prior work; both datasets have been used by previous studies [47, 101] to evaluate the tools we are comparing with. First, we use the CVE dataset, in which the vulnerabilities and their locations are known for each contract. Thus, we have the ground truth for our comparison. For each tool, we run the tool on the contracts in the dataset and then count how many vulnerabilities the tool detects. This allows us to estimate the *recall* of each tool. As Ethainter's source code is not publicly available, in our experiments, we use the online deployment of Ethainter [163] mentioned in their paper.

Second, as the CVE dataset has limited contracts, we use the SmartBugs-wild dataset to compare precision of the tools. SmartBugs-wild dataset has no ground truth of the vulnerabilities in each contract. We thus follow the approach of SPCon [101] to manually inspect a subset of vulnerabilities flagged by each tool in order to determine whether these vulnerabilities are true-positives or false-positives. For the tools that detect other classes of vulnerabilities, only the access control vulnerabilities are selected for inspection and for calculating the precision (which is a fraction of true access control vulnerabilities of all selected samples). This does not have an effect on the precision of these tools as the reported access control false-positives by these tools are not true vulnerabilities of other types detected by the tools.

To avoid bias, we focused on the subset of vulnerabilities that was selected by the authors of SPCon, and borrowed their inspection results for the tools we used for comparison. In that work, for each tool that reported a low number of vulnerabilities (Maian, Manticore, and SPCon), all the reported cases were inspected (44, 47, and 44, respectively). For all other tools, statistical sampling was used to obtain a 90% confidence level and a margin of error of 10% on whether the sample is representative of all reported cases. Thus, the number of samples selected is different for each tool. We applied the same approach to select samples

for *AChecker*: approximately 60 samples. Further, for *AChecker*, two smart contract security researchers independently inspected the vulnerabilities reported by the tool and only the vulnerabilities agreed on by both researchers are considered as true vulnerabilities. We use the inspection results to estimate the *precision* of each tool.

To answer **RQ2**, we run *AChecker* on the SmartBugs-wild dataset and randomly select a subset of 10 contracts marked by *AChecker* as having *potentially intended behaviors*; we add to this set 10 randomly selected contracts marked by *AChecker* as vulnerable. We recruited a third-party expert – a graduate student with smart contracts security auditing expertise, who is not an author of this paper. We asked the expert to manually inspect the contracts to determine which constitute true vulnerabilities, without knowing about our classification of intended behaviors. Our goal is to check if the results of the expert's classification of the intended behaviors match the results reported by *AChecker*. We also check whether the contracts we mark as potentially intended behaviors are also reported as vulnerabilities by prior work.

To avoid bias, we used the reported analysis results for Securify, Mythril, Manticore, SmartCheck, Maian, and Slither, and ran the other two tools, Ethainter and SPCon on these contracts ourselves (as prior work did not report these results).

To answer **RQ3**, we run *AChecker* on SmartBugs-wild and Popular-contracts datasets, and calculate the average analysis time of the contracts that *AChecker* analyzed successfully. When *AChecker* cannot analyze a contract within the timeout-value window, we consider *AChecker* as unable to analyze it.

To answer **RQ4**, we count contracts flagged as having access control vulnerabilities by *AChecker* on the SmartBugs-wild dataset and Popular-contracts dataset.

6.4.2 Experimental Results

RQ1 (Effectiveness)

CVE dataset Table 6.3 shows the results of comparing *AChecker* with the prior work on the CVE dataset. In the table, ✓denotes when the tool was able to detect the vulnerability for the corresponding CVE. The bottom row shows the overall

recall (detection rate) for each tool on the CVE dataset.

From the table, we see that AChecker *detected most of the vulnerabilities, and has a higher recall than all the other tools.* In particular, *AChecker* detected eight of the nine vulnerabilities detected by the other tools, and *four additional vulnerabilities* detected by none of the other tools (Section 6.5 discusses the reason for the three undetected vulnerabilities). Thus, *AChecker* has an overall **recall** of **80%** on the CVE dataset. SPCon had the next best recall (60%). The other tools all had less than 30% recall on the CVE dataset.

Table 6.3: Comparison with prior work on the CVE dataset (Recall).

CVE	Slither	Maian	SmartCheck	Manticore	Mythril	Securify2	Ethainter	SPCon	*AChecker*
CVE-2018-10666								✓	✓
CVE-2018-10705								✓	✓
CVE-2018-11329								✓	✓
CVE-2018-19830									✓
CVE-2018-19831					✓			✓	✓
CVE-2018-19832		✓			✓			✓	✓
CVE-2018-19833									✓
CVE-2018-19834									✓
CVE-2019-15078		✓			✓			✓	✓
CVE-2019-15079								✓	
CVE-2019-15080								✓	✓
CVE-2020-17753	✓		✓		✓				
CVE-2020-35962									
CVE-2021-34272								✓	✓
CVE-2021-34273									✓
Recall%	6	13	6	0	26	0	0	60	80

SmartBugs-wild dataset Table 6.4 shows a summary of the results of comparing *AChecker* with the other existing tools on the SmartBugs-wild dataset. The first row (Reported) shows the total number of vulnerabilities reported by the tool, the second row (Inspected) shows the number of sample vulnerabilities we manually inspected to estimate the precision of the tool, and finally the row at the bottom (Precision) shows the estimated precision of the tool.

Table 6.4: Comparison with prior work on the SmartBugs-Wild dataset (Precision).

	Slither	Maian	SmartCheck	Manticore	Mythril	Securify	SPCon	*AChecker*
Reported	2,356	44	384	47	1,076	614	44	624
Inspected	66	44	58	47	64	61	44	60
Precision%	24	61	29	19	39	33	81.8	88.3

From Table 6.4, the results show that *our proposed approach,* AChecker*, has the highest* **precision (88.3%)** *among all the tools.* Further, the tools with the second best precision are SPCon (81.8%) and Maian (61.4%). However, we see from the table that SPCon and Maian reported a much smaller number of vulnerabilities, with 44 vulnerabilities reported by each tool in comparison to 624 vulnerabilities reported by *AChecker*. This could be an indicator that these tools missed a lot of true vulnerabilities, and hence they may have lower recall than what we have seen earlier on the CVE dataset.

To establish a better understanding of the capability of the prior work to detect existing vulnerabilities in the SmartBugs-wild dataset, we checked how many of the true vulnerabilities reported by *AChecker* (we have manually confirmed these as true vulnerabilities), were reported by each of the existing tools. To avoid bias, we have used the original analysis results available with SmartBugs-wild dataset [47] for six tools: Slither, Maian, SmartCheck, Manticore, Mythril, and Securify. We ourselves ran the other two tools (Ethainter and SPCon) that were not considered in the SmartBugs study [47].

The results are shown in Table 6.5. The results show that *out of the 53 true vulnerabilities that are detected by* AChecker*, all the other tools together detected only 24 vulnerabilities.* We also cross-checked the vulnerabilities reported by other tools from the total true positives. *AChecker* (69.62%) outperformed all tools in reporting true vulnerabilities: Slither (44.4%), Maian (15.8%), Smartcheck (15.8%), Manticore (5.9%), Mythril (43.3%), Securify (15.2%), Ethainer (7%), SPCon (21%).

Table 6.5: Comparison with prior work on the SmartBugs-wild dataset (Recall).

TPs reported by *AChecker*	Slither	Maian	SmartCheck	Manticore	Mythril	Securify	Ethainter	SPCon
53	11	6	0	0	11	5	0	10

Answer to RQ1: *AChecker* outperforms all existing tools for detecting access control vulnerabilities, and is able to detect more true vulnerabilities with fewer false alarms.

RQ2 (Potentially Intended Behaviors)

Table 6.6 shows the results of the manual inspection for the 20 selected contracts. The first row shows the results for contracts that are marked by *AChecker* as vulnerable and the second row shows the results for contracts that are marked as potentially intended behaviors. The results demonstrate that the third-party expert classified eight of the 10 vulnerable cases reported by *AChecker* as true vulnerabilities, and two as false-positives. One false-positive is for a code protected by an access control checked in another external contract; hence, not recognized by *AChecker*. In the second case, the contract allows anyone to update the contract manager, but it calls an external code to authenticate the new manager, and it may eventually store the authenticated manager – the external code is not available to verify, though.

Further, the third-party expert classified *all the 10 potentially intended behavior cases* reported by *AChecker* as benign cases intentionally implemented rather than vulnerabilities. Moreover, we found that 70% of the 10 intended behavior cases were reported by the existing tools as vulnerabilities, adding to the false-positives of these tools. Overall, *AChecker* reported 418 functions with intended behavior cases. 221 of these cases are also reported by other tools as having access control vulnerabilities (though we did not validate them). This highlights the need for an approach like *AChecker* that filters the intended behaviors from exploitable true vulnerabilities.

Table 6.6: Intended behavior filtering (Precision).

Classification	Samples	True Vulnerabilities
Vulnerable	10	8
Potentially intended behavior	10	0

Answer to RQ2: Our approach for filtering potentially intended behavior has high precision in inferring the true intended behavior cases.

RQ3 (Performance)

By running *AChecker* on all the contracts in the SmartBugs-wild dataset, we found that the average analysis time of *AChecker* per contract is 11.74 seconds. This time includes the time taken by the symbolic execution to filter potentially intended behaviors. We obtained a similar average analysis time, 10.55 seconds for *AChecker* when running on the Popular-contracts dataset. In both datasets, *AChecker* timed out in about 17% of the contracts. Most of the timeouts were due to the inability of teEther [17] to generate the CFG.

Answer to RQ3: *AChecker* has an average analysis time of about 11 seconds per contract.

RQ4 (Prevalence of Access Control Vulnerabilities)

SmartBugs-wild dataset As mentioned in the results of RQ1, *AChecker* flagged 624 contracts as having access control vulnerabilities with a about 88% precision.

Popular-contracts dataset Table 6.7 shows the results of analyzing the Popular-contracts dataset by *AChecker*. Results show that *AChecker* flagged 21 contracts as having access control vulnerabilities. To determine how many of these reported vulnerabilities are true vulnerabilities, we inspected all the reported cases in all the contracts whose source code is available on Etherscan [52], an explorer of the

Ethereum blockchain. We restricted ourselves to contracts with source code as it is difficult to inspect the bytecode of contracts for vulnerabilities. Further, without the source code, running a smart contract following the trace found by taint-flow and symbolic execution is insufficient to identify vulnerabilities. Out of the flagged 21 contracts, 10 contracts have source code available. Our manual inspection of the vulnerabilities in these 10 contracts show that 9 of the vulnerabilities are true vulnerabilities (90% precision), and one is false-positive.

Table 6.7: Results of analyzing Popular-contracts by *AChecker*.

Flagged	Inspected	TPs
21	10	9 (90%)

Answer to RQ4: *AChecker* flagged 624 and 21 vulnerable contracts in the SmartBugs-wild and Popular-contracts datasets, respectively, showing that access control vulnerabilities are prevalent in these datasets.

6.5 Discussion

6.5.1 Result Analysis

As mentioned in RQ1, *AChecker* did not report three vulnerabilities in the CVE dataset. In one case, `tx.origin` is used in an access control check instead of `msg.sender`. *AChecker* does not analyze for this case as it is an outdated vulnerability that is found mostly in old contracts and can be found using a string search. Of the existing static analysis tools, Mythril, Slither, and SmartCheck can detect this vulnerability. Adding support for `tx.origin` in *AChecker* is straightforward, and is a potential direction for future work.

The other two cases are for functions that lack access control. These two cases are out of the scope of *AChecker* as *AChecker* analyzes for missing access control for a set of pre-defined instructions. In general, static analysis approaches such as *AChecker* cannot decide the contract functions that should be protected, due to the

lack of access control specifications. Although SPCon can detect these two cases as it uses the transactions history to mine the access control policy of the contract, neither of these two cases is detected by SPCon.

The results of *AChecker* on the three datasets show that most of the reported vulnerabilities are caused due to having *violated access control checks*. Only two existing analysis tools, SPCon and Ethainter, analyze for these vulnerabilities. However, both tools failed to detect many of the vulnerabilities found by *AChecker*. Ethainter's dependency on specific syntactic patterns was the main reason for its undetected cases.

For SPCon, the contracts that can be analyzed are limited due to having a low number of transactions, and it is challenging to increase the number of transactions to improve SPCon. As discussed (Section 3.2.2), SPCon requires transactions to be performed by only authorized users of the functions, as per the desired contract security policy, and it is difficult to determine whether or not a transaction obeys the security policy when the contract security policy is not known. This is because transactions that do not obey the security policy would result in increased false-negatives and false-positives.

Moreover, SPCon failed to mine access control roles for several vulnerable contracts where each contract has tens of thousands of transactions. An example taken from a vulnerable contract having about 183,533 transactions [133] is shown in Figure 6.8. This contract has a vulnerability that enables anyone to become an owner of the contract and perform actions such as withdrawing the contract Ether balance. SPCon was not able to mine the access control role for the vulnerable function `SAC` because this function was called by multiple different accounts; hence SPCon's approach considers it as a public function that can be called by anyone, and ignores it.

```
1 function SAC() public {
2    owner = msg.sender;
3    balances[owner] = totalDistributed;
4  }
```

Figure 6.8: Violated access control check vulnerability undetected by prior analysis tools.

The results of RQ2 also highlighted the capability of *AChecker* to infer many cases of intended behavior reported as vulnerabilities by the existing analysis tools. An example case is shown in Figure 6.9. This code is taken from a contract implementing a token named Peculium [121], and there are 14,972 transactions on this contract at the time of writing. The design of the contract allows the freeze possibility of tokens. Thus, the `transfer` function (line 8) has an access control check at line 9 so that only unfrozen accounts (stored at `balancesCanSell`) can transfer tokens. The contract allows token owners to defrost their tokens after a predefined date using the function `defrostToken`. Analyzing constraints of the `defrostToken` function enabled *AChecker* to infer that this is a potential intended behavior of the contract.

```
1  mapping(address => bool) public balancesCanSell;
2  uint256 public dateDefrost;
3  function defrostToken() public
4  { // Defrost your tokens, after the date of the defrost
5    require(now>dateDefrost);
6    balancesCanSell[msg.sender]=true;
7  }
8  function transfer(address _to, uint256 _value) public returns (bool) {

9    require(balancesCanSell[msg.sender]);
10   return BasicToken.transfer(_to,_value);
11 }
```

Figure 6.9: Intended behavior reported as a vulnerability by prior work.

To summarize, two explicit assumptions have been made in our design of the proposed approach, *AChecker*, to detect access control vulnerabilities. The first assumption is that caller-based access control checks can be identified and distinguished from other non-access control checks using data-flow analysis. The second assumption is that non-access control checks handling manipulation of access control data are intentionally added by developers to allow manipulation of access control data. Our findings from the evaluation experiments supported the validity of both assumptions, outperforming prior work relying on predefined code patterns and the transactions history. The evaluation experiments did not find any access control checks that *AChecker* was not able to identify, and *AChecker* distinguished the intended behavior cases with high precision based on a manual inspection done by a third-party expert. However, there are no guarantees provided

by *AChecker* that the filtered potentially intended behaviors are, indeed, intended behaviors. Our goal of applying such heuristics is to help end-users of *AChecker* distinguish more certain vulnerabilities. Thus, end-users need to inspect the cases classified as potentially intended behaviors.

6.5.2 Limitations

AChecker has three main limitations. First, it performs intra-contract analysis and hence does not analyze access control delegated to other contracts. Second, it analyzes missing access control only for a set of predefined critical instructions due to the lack of access control specifications of contracts. Thus, *AChecker* does not analyze missing access control for functions that should be protected if they do not have predefined instructions. Finally, *AChecker* only checks access granted to the contract callers on functions and, what we call, critical instructions. This is a common way to authorize users in smart contracts, but other methods are also possible.

6.5.3 Threats to Validity.

An external threat to validity is the limited number of contracts used to evaluate effectiveness of *AChecker* and compare it to the prior work (15 contracts). We partially mitigated this threat by using a dataset with a confirmed set of CVEs. In addition, we used the Smartbugs dataset consisting of 47,518 real-world contracts, and the Popular-contracts dataset (with 3,000 real-world widely used contracts) to evaluate the precision of *AChecker* and compare with the prior work. Both datasets have been used by previous studies to evaluate the analysis tools we use for comparison. We avoided using datasets injected with artificial vulnerabilities as, to our knowledge, there is no existing tool that injects access control vulnerabilities covered by the proposed approach - injecting such vulnerabilities ourselves may bias the evaluation.

An internal threat to validity is the possible bias due to manual inspection of the sample vulnerabilities to compare with the prior work. We partially mitigated this threat in two ways. First, we used the inspection results from previous studies for the analysis tools we are comparing with *AChecker*, where possible. Second, two

smart contract researchers independently inspected the vulnerabilities of *AChecker*, and only those agreed on by both are counted as true vulnerabilities.

6.6 Summary

Access control vulnerabilities in smart contracts can lead to significant financial loss, and occur due to either violated or missing access control checks. This chapter proposed *AChecker*, a static analysis approach for finding violated and missed access control checks in smart contracts. *AChecker* employs a data-flow analysis technique to determine access control checks implemented in the contract, and a symbolic execution-based approach to distinguish cases implemented as intended behavior from vulnerabilities. We evaluated *AChecker* on three smart contract datasets, and the results show that *AChecker* effectively discovered access control vulnerabilities with a much higher recall and precision than eight existing analysis tools. Further, *AChecker* flagged 21 contracts of the most popular Ethereum contracts as having access control vulnerabilities - 90% of these were true vulnerabilities. Finally, *AChecker* has an average analysis time of 11 seconds per contract.

Chapter 7

Conclusion

In this chapter, we first provide a summary of the dissertation and discuss its anticipated impact. Following this, we discuss the research limitations and briefly outline potential directions for future research work.

7.1 Summary

The overarching goal of this dissertation is to build static analysis approaches for finding gas-related and access control vulnerabilities in smart contracts. To achieve this, we introduced three research questions in Chapter 1, which were addressed in Chapters 4, 5, and 6. We first wanted to understand the efficacy of the current smart contract static analysis tools in detecting vulnerabilities. The understanding then helped us improve the state of the static analysis for detecting vulnerabilities in smart contracts. We targeted the detection of two main categories of vulnerabilities common in smart contracts, i.e., gas-related and access control vulnerabilities, as summarized in Table 7.1. These two categories involve different classes of vulnerabilities. Further, the targeted vulnerability classes are all critical and have significant risks, e.g., Ether loss, DoS attacks on smart contracts, and control hijacking of smart contracts. In what follows, we summarize the three main parts of the dissertation that addressed the research questions.

RQ1. *How effective are smart contract static analysis tools in detecting security bugs?*

In Chapter 4, we studied the different security bugs occurring in smart contracts, and we wanted to understand the strengths and weaknesses of the current static analysis tools in detecting these security bugs. Due to the lack of a systematic evaluation approach for smart contract analysis tools, we started by manually injecting security bugs into smart contract Solidity code, and we found the process tedious, time-consuming, and error-prone. To address this, we found that injecting security bugs into the code can be done systematically and in an automated way. Based on this observation, we proposed the *SolidiFI* approach, which systematically injects security bugs into smart contract Solidity source code at the AST level and then uses the generated buggy code to evaluate the false-negatives and false-positive of the static analysis tools under study. Using *SolidiFI*, we evaluated well-known static analysis tools. The evaluation results showed that these tools fail to detect even simple cases of security bugs that they are supposed to detect, and report massive false-positives.

RQ2. *How can we effectively detect gas-related vulnerabilities in smart contracts?*

In Chapter 5, this dissertation targeted finding gas-related vulnerabilities and code smells in smart contracts. These types of vulnerabilities and code smells are common in smart contracts, leading to unwanted behaviors in vulnerable smart contracts enforced by malicious attackers, such as DoS attacks. Table 7.1 summarizes the notable forms and potential risks of the targeted gas-related vulnerabilities. We found that the other techniques rely on specific and pre-defined code patterns to detect gas-related vulnerabilities, which results in several false-negatives and false-positives. Our insight in this chapter is that the dependency of specific critical code statements on data items manipulated by contract users causes gas-related vulnerabilities and code smells. This insight enabled us to formulate the detection of gas-related vulnerabilities and code smells as a taint analysis problem. Based on this, we proposed *eTainter*, an inter-procedural static taint analysis approach that tracks taint in the contracts' low-level EVM bytecode. *eTainter* considers smart contract peculiarities, such as tracking taint through the contract's persistent storage, and applies domain-specific optimizations to reduce false-positives. The evaluation results showed that our proposed approach achieves high precision and recall greater than 90%, significantly outperforming those of the prior work.

Table 7.1: A summary of the vulnerability classes targeted in this thesis. GR= gas-related vulnerability, AC= access control vulnerability.

Type	Name	Notable Forms	Potential Risks
GR	Unbounded Loops	Loops bounded by dynamic data items growing over time	DoS & Ether lockout
GR	DoS with Failed Call	An external call is in a loop's body, and the call reverts execution in case of failure	DoS & Ether lockout
GR	Unhandled External Calls	Failures of an external call are not handled	DoS & data inconsistencies leading to unwanted behaviors
GR	Hardcoded Gas Calls	An external call that limits gas assigned for the call execution	DoS & Ether lockout
AC	Violated Access Control	Ability of unauthorized contract users to manipulate state variables storing access control data	Contract control hijacking by attackers, stealing Ether, destroying contracts, etc.
AC	Missing Access Control	Ability of unauthorized contracts users to either execute critical instructions or manipulate arguments of the critical instructions	Stealing Ether, destroying contracts, etc.

RQ3. *How can we effectively detect access control vulnerabilities in smart contracts with no contract access control specifications?*

In Chapter 6, we targeted the detection of access control vulnerabilities in smart contracts. A summary of the targeted access control vulnerabilities and their potential risks is shown in Table 7.1. The two main challenges for finding access control vulnerabilities in smart contracts are identifying access control checks implemented in the contracts and distinguishing contract-intended functionality from vulnerabilities in cases where unauthorized users can manipulate access control data under non-access control constraints. In this chapter, we came up with two insights. The first insight is that access control checks have unique functionality that can be inferred by analyzing the data flows in the contract code. The second insight is that the non-access control constraints implemented by developers to allow manipulation of access control data (when satisfied) can be precisely inferred using symbolic execution. Using these insights, we proposed a static data-flow

analysis technique to identify access control checks to detect violated and missing access control in the smart contract. Further, we proposed a symbolic-based analysis method to infer cases where the developer intentionally allows the manipulation of the access control data under non-access control constraints. We built a consolidated approach, *AChecker*, combining our proposed techniques to detect both violated and missing access control based on taint tracking. Our proposed approach's effectiveness (80% recall and 88% precision) exceeded those of all related tools finding access control vulnerabilities and rely on pre-defined code patterns and transactions history.

7.2 Dissertation Impact

We expect the dissertation to have the following impact.

The dissertation highlights the limitations of the current static analysis tools for smart contracts, and it shows that existing tools are still in the infancy as they report high false alarms and miss detection of many vulnerabilities. Our evaluation study of the effectiveness of static analysis tools in detecting security bugs provides insights for security researchers and tool developers that can be leveraged to propose solid static analysis-based approaches for finding vulnerabilities in smart contracts. These approaches can be designed to overcome the limitations of existing tools and to enhance their effectiveness in detecting vulnerabilities in smart contracts. Furthermore, our work on evaluating static analysis tools and our proposed approach, *SolidiFI*, motivated several follow-up research efforts based on this idea [10, 43, 55, 84, 97, 126, 129, 176]. Further, our approach *SolidiFI* was used in several research studies to evaluate more analysis tools and to generate benchmarks to evaluate the effectiveness of newly proposed analysis approaches for smart contracts.

Furthermore, unlike traditional programs, smart contracts have a set of peculiarities that make static analysis a challenging process. For instance, each contract has access to a storage area on the blockchain where the contract can store persistent data maintained over executions. In addition, EVM uses a cryptographic hashing-based method to locate and reference dynamic data items (e.g., arrays) in the contract storage. In this dissertation, we advance the state of the static analysis,

specifically, static taint and data-flow analysis of the EVM bytecode, to address those challenges. Our proposed static taint and data-flow analysis approaches can be adopted to analyze Ethereum smart contracts and smart contracts supported by other blockchains that are EVM compatible and compile to EVM bytecode.

For finding vulnerabilities in smart contracts, the dissertation highlights the limitations of the current static analysis tools and provides several insights for detecting vulnerabilities. We believe that the insights from this dissertation will form a basis for building more effective static analysis approaches in the future. Further, the proposed vulnerability detection techniques in this dissertation can be integrated into the smart contracts development process to help developers detect potential vulnerabilities early during the development phase.

More specifically, the dissertation shows that most of the current static analysis tools typically rely on specific and ad-hoc patterns to analyze contracts for vulnerabilities rather than analyzing contracts for the root causes leading to vulnerabilities, resulting in high false-negatives and false-positives. We demonstrated in Chapters 5 and 6 that vulnerability detection solutions, capable of detecting vulnerabilities with high recall and precision, can be built by mainly identifying critical parts of the contract code that can form vulnerabilities and defining them as security properties. Then the contract code is analyzed for potential violations of such security properties. This insight formed the foundation for our proposed vulnerability detection approaches in this dissertation. Our approach *eTainter*, for example, focuses on identifying gas exceptions triggered by data items provided or manipulated by contract users, which is a common root cause of gas-related vulnerabilities. Similarly, our approach *AChecker* first infers access control checks implemented in contracts by identifying unique functionality in access control checks that compare callers of the contracts against specific access control state variables. It then analyzes for missing and violated access control checks resulting in access control vulnerabilities.

Overall, the dissertation highlights the limitations of the current static analysis tools and the challenges of building static analysis tools for smart contracts having high precision and recall. The dissertation also proposes approaches to improve the state of static analysis for smart contracts. By improving the effectiveness of static tools, developers can more confidently identify and address potential vul-

nerabilities in smart contracts before their deployment, thereby reducing the risks of financial loss and other negative consequences, and protecting the integrity of blockchain systems.

7.3 Limitations

In what follows, we discuss the limitations of the approaches developed in this research and the limitations inherent to the field of smart contracts related to vulnerability detection research.

7.3.1 Limitations of the approaches developed in this thesis

Our research conducted in this thesis has the following limitations.

Not covering all possible security bugs introduced by developers. In our proposed approach for evaluating static analysis tools, *SolidiFI*, some vulnerabilities the developers introduce in smart contracts may not be the same as vulnerabilities injected by *SolidiFI*. Therefore, the effectiveness of the static analysis tools in detecting vulnerabilities we measured may not be similar to the effectiveness of these tools in catching vulnerabilities introduced by developers.

Intra-contract analysis-based approaches. In this research, we proposed intra-contract analysis-based approaches. Intra-contract analysis may not be complete and sound, as smart contracts may delegate part of their functionality to other external contracts not involved in the analysis. This can result in false-negatives and false-positives, as has been discussed in Chapter 6.

Caller-based access control. Our research targeted access control vulnerabilities by analyzing access control granted to contract callers. This is a common way to authorize users in smart contracts. However, in some cases, smart contracts may use other forms of access control, e.g., based on time. Vulnerabilities due to improper implementation of these forms of access control are thus out of the scope of our analysis. Not covering other forms of access control could result in false-negatives or false-positives even though we did not see any such cases in our evaluation experiments.

7.3.2 Limitations inherent to the smart contracts field

In addition to the research limitations discussed above, there are other limitations related to the broad area of smart contracts in general, affecting the research targeting vulnerability detection in smart contracts.

Lack of standard datasets. The smart contracts field lacks standard datasets of vulnerable contracts having a ground truth to be used in research studies targeting vulnerability detection. In this research, we tried to mitigate this limitation by borrowing datasets used by prior work in our evaluation experiments (i.e., MadMax and SmartBugs-wild datasets), especially to compare our proposed approaches with the prior work. However, datasets from prior work also have limitations (e.g., the MadMax dataset has a limited number of contracts, and most of the contracts in the SmartBugs-wild dataset are small in size). Thus, we needed to collect other datasets from the contracts deployed on the Ethereum blockchain. This might introduce some bias in our evaluation of the proposed approaches even though we tried to avoid bias by taking a complete snapshot of the blockchain (e.g., Ethereum dataset) and collecting contracts with a large number of transactions on the blockchain (e.g., Popular-contracts dataset).

Further, in our research, we needed to spend more effort validating vulnerabilities by manually inspecting smart contracts to evaluate the effectiveness of the analysis tools. Manual inspection to identify vulnerabilities is based on the inspectors' understanding of the functionality of the smart contracts. The manual inspection could have introduced bias in our evaluation results.

Additionally, in the field of smart contracts, there is a lack of standard code datasets, e.g., GitHub repositories, for results validation. In our experiments, we collected the contract datasets from the blockchain. As a result, due to blockchain anonymity, we could not reach out to the developers to validate the findings of our vulnerability detection approaches.

Limited analysis tools. In the field of smart contracts, there is a lack of reliable analysis tools that can produce the CFG and SSA of the code. The present tools are incapable of creating the CFG and SSA for many contracts. To automate our proposed vulnerability analysis techniques, we had to rely on these tools to generate the CFG and SSA of the analyzed contracts. The dependency on these tools

resulted in many contracts that could not be analyzed by our automated methods during the evaluation experiments, therefore possibly underestimating the prevalence of the gas-related and access control vulnerabilities we studied in this thesis.

7.4 Future Research Directions

In this dissertation, our research aimed to develop static analysis approaches for vulnerabilities in smart contracts, to improve the overall security of smart contracts, and help smart contract developers write more secure smart contracts. Our proposed approaches focused on understanding the effectiveness of static analysis tools and detecting gas-related and access control vulnerabilities. However, there is still much left to be addressed in this area. In the following, we suggest several potential research directions based on the findings and contributions of this research.

Safe Languages. Many security issues in smart contracts stem from how the Solidity programming language was designed. For example, the lack of a permission-based security model in Solidity forces developers to implement their custom access control checks, which creates several vulnerabilities and makes it challenging to reason about whether the contract code satisfies access control requirements. One of the potential future directions to enhance the security of smart contracts is to build better and safe programming languages that specifically address the security and safety needs of smart contracts in their design.

Standardization. Smart contracts work on the top of the blockchain, and their functionality differs from traditional software applications. Developing secure smart contracts, therefore, requires developers' mastery of the underlying blockchain technology and how smart contracts work. For instance, smart contracts execution is deterministic, and implementing randomization and data feeds from external sources (outside the blockchain), such as price feeds, requires deep knowledge of blockchain and the smart contracts ecosystem. Hence, there is a need for standardization efforts. Standardization can be instrumental to bridge this knowledge gap by building development frameworks backed with standard libraries to facilitate the implementation of secure smart contracts. Although some standards for smart contracts have been developed, such as libraries and data ora-

cles [116], the scope of those standards remains limited and scattered, and there is a lack of development frameworks that consolidate those standards.

Focusing on Root Causes of Vulnerabilities. As demonstrated through this dissertation, a major and common drawback of current smart contract analysis tools is that tools rely on analyzing code for specific known vulnerable patterns without taking into account the root causes of the vulnerabilities being analyzed. In this dissertation, we focused on analyzing contract code for root causes leading to gas-related and access control vulnerabilities, and as presented in Chapters 5 and 6, results showed a significant improvement in the effectiveness of detecting targeted vulnerabilities compared with the prior work that mainly relied on ad-hoc code patterns. One promising direction is to propose approaches that generalize the idea to larger classes of vulnerabilities by focusing on the root causes of the vulnerabilities.

Cross-contract Analysis. The natural functionality of smart contracts requires interacting with other external smart contracts. In addition, to enhance the security and reusability of code, certain functionality of smart contracts gets delegated to other external contracts, for instance, to check access control. Therefore, effective analysis of smart contracts for vulnerabilities would be improved by performing inter-contract (cross-contract) analysis. We show in Chapter 6 that intra-contract analysis alone can result in some false-negatives and false-positives when smart contracts delegate access control checks to other external contracts. In these cases, our current approach *AChecker* will not be able to analyze such delegations and infer delegated access control checks. One research direction is to expand our static analysis approaches and build other cross-contract analysis approaches.

Fully-Automated Evaluation Approach of Analysis Tools. As discussed in Chapter 4, our proposed methodology for evaluating static analysis tools, *SolidiFI*, employs three different approaches to introduce security bugs in the source code of smart contracts and generate buggy contracts. However, most of the security bugs get injected through the 'full code snippet' approach that relies on manually prepared code snippets, as the other two approaches, 'code transformation' and 'weakening security mechanisms,' apply only to a few bug types that form specific syntax patterns. A potential research direction is to expand the evaluation methodology, *SolidiFI*, and focus on automation of the code snippet generation by

analyzing the contract code. This would also improve the quality of the injected security bugs to be more natural and related to the contract code.

Bibliography

[1] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania. Astraea: A decentralized blockchain oracle. In *2018 IEEE international conference on internet of things (IThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*, pages 1145–1152. IEEE, 2018. → page 34

[2] S. Akca, A. Rajan, and C. Peng. Solanalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489. IEEE, 2019. → page 26

[3] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *International Symposium on Automated Technology for Verification and Analysis*, pages 513–520. Springer, 2018. → pages 31, 35

[4] S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77. ACM, 2018. → page 33

[5] I. Ashraf, X. Ma, B. Jiang, and W. Chan. Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. *IEEE Access*, 8:99552–99564, 2020. doi:https://doi.org/10.1109/ACCESS.2020.2995183. → page 29

[6] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*, pages 164–186. Springer, 2017. → page 47

[7] S. Azzopardi, J. Ellul, and G. J. Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. In *International Conference on Runtime Verification*, pages 113–137. Springer, 2018. → page 32

[8] BAFC. Business alliance financial circle (BAFC) token. URL https://etherscan.io/address/0x9924A7E3A2756Ab8B9A828485f052b6693AaA33E. → page 107

[9] T. Bahrynovska. History of ethereum security vulnerabilities, hacks, and their fixes, 2017. URL https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes. → pages 44, 45, 54

[10] S. Behan. *Solidity Smart Contract Testing with Static Analysis Tools*. PhD thesis, Dublin, National College of Ireland, 2022. → page 136

[11] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016. → page 33

[12] Bitcoin-Exchange-Guide-News-Team. The parity wallet breach, 2017. URL https://bitcoinexchangeguide.com/parity-wallet-breach. → page 36

[13] C. Blockchain. Bamboo: a language for morphing smart contracts. URL https://github.com/pirapira/bamboo. → page 34

[14] F. Bond. eth-mutants, 2018. URL https://github.com/federicobond/eth-mutants. → page 27

[15] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1263–1280, 2018. → page 25

[16] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018. → page 40

[17] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 454–469, 2020. → pages 5, 7, 29, 30, 31, 35, 106, 107, 109, 122, 127

[18] R. Browne. Accidental's bug froze $280 million worth of ether in parity wallet, 2018. URL https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html. → pages 7, 105

[19] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform. 7, 2014. URL https://ethereum.org/en/whitepaper. → page 12

[20] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013. → page 3

[21] Chainlink. Top 6 smart contract languages in 2023. URL https://chain.link/education-hub/smart-contract-programming-languages. → page 12

[22] H. Chen, M. Pendleton, L. Njilla, and S. Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020. → page 34

[23] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446. IEEE, 2017. doi:https://doi.org/10.1109/SANER.2017.7884650. → pages 29, 35

[24] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *Information Security Practice and Experience: 13th International Conference, ISPEC 2017, Melbourne, VIC, Australia, December 13–15, 2017, Proceedings 13*, pages 3–24. Springer, 2017. → page 34

[25] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pages 81–84. IEEE, 2018. → pages 29, 35

[26] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 2020. doi:https://doi.org/10.1109/TETC.2020.2979019. → pages 29, 35

[27] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. 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. IEEE, 2021. doi:https://doi.org/10.1109/ASE51524.2021.9678888. → pages 3, 32, 35

[28] C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016. → page 3

[29] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009. → page 24

[30] CoinDesk. Ethereum's istanbul upgrade will break 680 smart contracts on aragon. URL https://www.coindesk.com/ethereums-istanbul-upgrade-will-break-680-smart-contracts-on-aragon. → page 80

[31] ConsenSys. Don't use transfer() or send(), . URL https://consensys.github.io/smart-contract-best-practices/development-recommendations/general/external-calls/#dont-use-transfer-or-send. → page 80

[32] ConsenSys. Stop using solidity's transfer() now, . URL https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/. → page 80

[33] ConsenSys. Ethereum smart contract best practices, . URL https://consensys.github.io/smart-contract-best-practices/attacks. → page 75

[34] ConsenSys. Mythril, 2019. URL https://github.com/ConsenSys/mythril. → pages 5, 29, 31, 35, 106, 107, 122

[35] ConsenSys. Infura, 2020. URL https://infura.io. → page 59

[36] ConsenSys. Metamask: A crypto wallet & gateway to blockchain apps, 2020. URL https://metamask.io. → page 59

[37] T. Cook, A. Latham, and J. H. Lee. Dappguard: Active monitoring and defense for solidity smart contracts. → page 32

[38] Crytic. Echdina. URL https://github.com/crytic/echidna. → pages 32, 35

[39] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, 1989. → page 85

[40] P. Daian. Analysis of the dao exploit, 2016. URL https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit. → pages 2, 36, 46, 54

[41] C. Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Springer, 2017. → pages 1, 37

[42] DefiLlama. Defi tvl breakdown by smart contract languages. URL https://defillama.com/languages. → page 12

[43] B. Dia, N. Ivaki, and N. Laranjeiro. An empirical evaluation of the effectiveness of smart contract verification tools. In *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 17–26. IEEE, 2021. → pages 27, 136

[44] S. documentation. Gas limit and loops, 2022. URL https://docs.soliditylang.org/en/v0.5.11/security-considerations.html#gas-limit-and-loops. → page 76

[45] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016. → page 26

[46] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. *arXiv preprint arXiv:1910.10601*, 2019. → page 26

[47] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541, 2020. → pages 37, 91, 106, 121, 122, 125

[48] M. Eshghie, C. Artho, and D. Gurov. Dynamic vulnerability detection on smart contracts using machine learning. In *Evaluation and Assessment in Software Engineering*, pages 305–312. 2021. → page 32

[49] Ethereum. Networks. URL https://ethereum.org/en/developers/docs/networks. → page 1

[50] Ethereum. The solidity contract-oriented programming language, 2015. URL https://github.com/ethereum/solidity. → page 12

[51] Etherscan. Airdrop contract, . URL https://etherscan.io/address/0x220348263aab5a038845483f6096895aa59f3977. → page 110

[52] Etherscan. Etherscan, . URL https://etherscan.io. → pages 52, 127

[53] Etherscan. Pipot contract, 2021. URL https://etherscan.io/address/0x14d01b02d1a2aa051082810d77f8d64c80937cd5#code. → page 72

[54] Ethervm. Ethereum virtual machine opcodes. URL https://ethervm.io. → pages xi, 22

[55] G. Faura, C. Siewiersky, and I. Tal. A user-centric evaluation of smart contract analysis tools in decentralised finance (defi). In *Proceedings of the International Conference on Cybersecurity, Situational Awareness and Social Media: Cyber Science 2022; 20–21 June; Wales*, pages 453–476. Springer, 2023. → pages 27, 136

[56] J. Feist, G. Grieco, and A. Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019. → pages 5, 31, 37, 40, 52, 106, 107, 122

[57] Y. Feng, E. Torlak, and R. Bodík. Precise attack synthesis for smart contracts. *CoRR*, abs/1902.06067, 2019. URL http://arxiv.org/abs/1902.06067. → pages 32, 35, 65

[58] Y. Feng, E. Torlak, and R. Bodik. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067*, 2019. → pages 31, 35

[59] K. Framework. K framework. URL http://www.kframework.org/index.php/Main_Page. → page 33

[60] J. Frank, C. Aschermann, and T. Holz. ETHBMC: A bounded model checker for smart contracts. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2757–2774, 2020. → pages 33, 35

[61] FStarLang. F*: A higher-order effectful language designed for program verification. URL https://www.fstar-lang.org. → page 33

[62] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, 47(12):2874–2891, 2020. → page 33

[63] A. Ghaleb. Towards effective static analysis approaches for security vulnerabilities in smart contracts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022. → page 9

[64] A. Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427, 2020. doi:https://doi.org/10.1145/3395363.3397385. → pages 5, 8, 32

[65] A. Ghaleb and K. Pattabiraman. SolidiFI, 2020. URL https://github.com/DependableSystemsLab/SolidiFI. → pages 11, 51

[66] A. Ghaleb and K. Pattabiraman. SolidiFI benchmark, 2020. URL https://github.com/DependableSystemsLab/SolidiFI-benchmark. → pages 11, 52

[67] A. Ghaleb, J. Rubin, and Pattabiraman. AChecker, 2022. URL https://github.com/DependableSystemsLab/AChecker. → pages 11, 107

[68] A. Ghaleb, J. Rubin, and Pattabiraman. eTainter artifact, 2022. URL https://doi.org/10.5281/zenodo.6578840. → page 11

[69] A. Ghaleb, J. Rubin, and Pattabiraman. eTainter, 2022. URL https://github.com/DependableSystemsLab/eTainter. → pages 11, 71

[70] A. Ghaleb, J. Rubin, and K. Pattabiraman. eTainter: Detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 728–739, 2022. → pages 6, 9, 106, 120, 121

[71] A. Ghaleb, J. Rubin, and K. Pattabiraman. AChecker artifact, 2023. URL https://doi.org/10.5281/zenodo.7626567. → page 11

[72] A. Ghaleb, J. Rubin, and K. Pattabiraman. AChecker: Statically detecting smart contract access control vulnerabilities. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pages 945–956, 2023. doi:10.1109/ICSE48619.2023.00087. → pages 7, 9

[73] Github. Madmax, 2018. URL https://github.com/nevillegrech/MadMax. → pages 73, 91, 94

[74] Github. rattle, 2018. URL https://github.com/crytic/rattle. → pages 85, 99

[75] Github. Ethereum etl, 2020. URL https://github.com/blockchain-etl/ethereum-etl. → page 90

[76] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):116, 2018. → pages 5, 6, 15, 17, 21, 28, 35, 76, 90, 102

[77] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Analyzing the out-of-gas world of smart contracts. *Communications of the ACM*, 63(10):87–95, 2020. doi:https://doi.org/10.1145/3416262. → page 90

[78] I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018. → pages 1, 12, 33

[79] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages*, 2(POPL):48, 2017. → page 32

[80] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002. → page 3

[81] M. S. Hecht. *Flow analysis of computer programs*. Elsevier Science Inc., 1977. → page 105

[82] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical report, 2017. → pages 33, 34

[83] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017. → page 33

[84] T. Hu, J. Li, A. Storhaug, and B. Li. Why smart contracts reported as vulnerable were not exploited? 2023. → page 136

[85] E. IO. Eos. io technical white paper. *EOS. IO (accessed 18 December 2017) https://github. com/EOSIO/Documentation*, page 9, 2017. → page 1

[86] Isabelle. Isabelle. URL https://isabelle.in.tum.de. → page 33

[87] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009. → page 24

[88] S. Ji, J. Wu, J. Qiu, and J. Dong. Effuzz: Efficient fuzzing by directed search for smart contracts. *Information and Software Technology*, page 107213, 2023. → page 32

[89] B. Jiang, Y. Liu, and W. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269. ACM, 2018. → pages 32, 35

[90] D. Johnston, S. O. Yilmaz, J. Kandah, N. Bentenitis, F. Hashemi, R. Gross, S. Wilkinson, and S. Mason. The general theory of decentralized applications, dapps. 2014. URL https://github.com/DavidJohnstonCEO/DecentralizedApplications. → page 1

[91] C. Kaner, J. Bach, and B. Pettichord. *Lessons learned in software testing*. John Wiley & Sons, 2008. → page 3

[92] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. → page 119

[93] kingoftheether.com. Post-mortem investigation. URL https://www.kingoftheether.com/postmortem.html. → page 78

[94] J. Kongmanee, P. Kijsanayothin, and R. Hewett. Securing smart contracts in blockchain. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 69–76. IEEE, 2019. → page 33

[95] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016. → page 34

[96] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium*

({USENIX} Security 18), pages 1317–1333. {USENIX Association}, 2018. → pages 17, 21, 29, 31, 35, 82, 85, 99, 120

[97] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 2022. → pages 27, 136

[98] S. Lagouvardos, N. Grech, I. Tsatiris, and Y. Smaragdakis. Precise static modeling of ethereum "memory". *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020. doi:https://doi.org/10.1145/3428258. → page 88

[99] A. Li, J. A. Choi, and F. Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 2020. → page 3

[100] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 65–68, 2018. → page 32

[101] Y. Liu, Y. Li, S.-W. Lin, and C. Artho. Finding permission bugs in smart contracts with role mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 716–727, 2022. → pages 5, 7, 29, 30, 35, 106, 107, 109, 121, 122

[102] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Oyente - an analysis tool for smart contracts, version 0.2.7. URL https://github.com/melonproject/oyente. → page 52

[103] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016. → pages 5, 31, 35, 37, 40, 41, 52, 62

[104] F. Mathieu and R. Mathee. Blocktix: decentralized event hosting and ticket distribution network, 2017. URL https://www.cryptoground.com/storage/files/1527588859-blocktix-wp-draft.pdf. → page 36

[105] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore. URL https://github.com/trailofbits/manticore. → pages 31, 52

[106] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019. → pages 5, 29, 35, 37, 40, 52, 106, 107, 122

[107] B. Mueller. Mythril. URL https://github.com/ConsenSys/mythril. → page 52

[108] B. Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 2018. → pages 31, 32, 37, 40, 41, 52

[109] S. Nakamoto. Bitcoin whitepaper. *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)*, 2008. → page 1

[110] B. Nassirzadeh, H. Sun, S. Banescu, and V. Ganesh. Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities. *arXiv preprint arXiv:2112.14771*, 2021. → pages 28, 35

[111] Z. Nehai, P.-Y. Piriou, and F. Daumas. Model-checking of smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 980–987. IEEE, 2018. → pages 3, 33, 34

[112] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. 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, 2020. → pages 32, 35

[113] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663. ACM, 2018. → pages 5, 29, 30, 31, 35, 106, 107, 122

[114] NVD. Cve-2018-10299 detail, 2020. URL https://nvd.nist.gov/vuln/detail/CVE-2018-10299. → page 45

[115] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century*, pages 34–44, 2001. → page 23

[116] OpenZeppelin. Openzeppelin contracts. URL https://github.com/OpenZeppelin/openzeppelin-contracts. → page 141

[117] M. Ouimet and K. Lundqvist. Formal software verification: Model checking and theorem proving. *Embedded Systems Laboratory Technical Report ESL-TIK-00214, Cambridge USA*, 2007. → page 24

[118] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 103–113. IBM Corp., 2018. → pages 37, 40

[119] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 912–915. ACM, 2018. → page 33

[120] PeckShield. New batchoverflow bug in multiple erc20 smart contracts (cve-2018–10299), 2018. URL https://peckshield.medium.com/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536. → page 54

[121] Peculium. Peculium (pcl) token. URL https://etherscan.io/address/0x53148bb4551707edf51a1e8d7a93698d18931225. → page 130

[122] C. Peng, S. Akca, and A. Rajan. Sif: A framework for solidity contract instrumentation and analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–473. IEEE, 2019. → page 26

[123] D. Perez and B. Livshits. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710*, 2019. → page 40

[124] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020. → pages 31, 35

[125] J. Pewny and T. Holz. Evilcoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 214–225. ACM, 2016. → pages 23, 25

[126] H. Rameder. Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools. 2021. → page 136

[127] REKT. Value defi - rekt 2, 2021. URL https://rekt.news/value-rekt2. → pages 2, 105

[128] Remix. Solidity IDE, 2017. URL http://remix.ethereum.org. → page 59

[129] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai. Empirical evaluation of smart contract testing: What is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 566–579, 2021. → pages 27, 136

[130] M. Rodler, W. Li, G. O. Karame, and L. Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934*, 2018. → page 29

[131] G. Rosu. An overview of the k semantic framework. Technical report, 2010. → page 33

[132] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. doi:10.1109/JSAC.2002.806121. → pages 6, 23, 70, 105

[133] Sachio. Sachio (sch) contract. URL https://etherscan.io/address/0xf34839b310097fcb4cf3a302dda8cc9b57501083. → page 129

[134] F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing safe smart contracts in flint. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, pages 218–219, 2018. → page 34

[135] SigmaPrime. solidity-security-blog, 2020. URL https://github.com/sigp/solidity-security-blog. → pages 45, 46

[136] Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, and I. Tsatiris. Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485540. → pages 30, 31, 35, 101

[137] SmartContractSecurity. Message call with hardcoded gas amount, . URL https://swcregistry.io/docs/SWC-134#hardcoded-gas-limitsso. → page 80

[138] SmartContractSecurity. Unprotected ether withdrawal, . URL https://swcregistry.io/docs/SWC-105. → page 109

[139] SmartContractSecurity. Unprotected selfdestruct instruction, . URL https://swcregistry.io/docs/SWC-106.

[140] SmartContractSecurity. Delegatecall to untrusted callee, . URL https://swcregistry.io/docs/SWC-112. → page 109

[141] SmartContractSecurity. Smart contract weakness classification and test cases, . URL https://swcregistry.io. → page 75

[142] Solidity. Security considerations, . URL https://docs.soliditylang.org/en/v0.5.11/security-considerations.html#security-considerations. → page 76

[143] Solidity. Re-entrancy, . URL https://docs.soliditylang.org/en/v0.5.11/security-considerations.html#re-entrancy. → page 80

[144] C. Staff. Ethereum classic (etc): A rift in the blockchain community. URL https://www.gemini.com/cryptopedia/ethereum-classic-etc-vs-eth. → page 2

[145] Sugoi. Sugoi nft nyc 2022. URL https://etherscan.io/address/0x8088f4612eadb9d60d5c8abf4a9d0fdfc3df2f1e. → page 110

[146] M. Swende. Eip-1884: Repricing for trie-size-dependent opcodes. URL https://eips.ethereum.org/EIPS/eip-1884. → page 80

[147] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu, et al. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. ACM, 2012. → page 37

[148] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck - a static analysis tool that detects vulnerabilities and bugs in solidity programs. URL https://github.com/smartdec/smartcheck. → page 52

[149] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. 2018. → pages 5, 29, 30, 35, 37, 40, 41, 52, 106, 107, 122

[150] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54 (7):1–38, 2021. doi:10.1145/3464421. → pages 3, 4, 32, 34

[151] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018. doi:10.1145/3274694.3274737. → pages 29, 31, 35

[152] Trailofbits. Slither, the solidity source analyzer. URL https://github.com/crytic/slither. → page 52

[153] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify. URL https://github.com/eth-sri/securify. → page 52

[154] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018. → pages 5, 29, 30, 35, 37, 40, 41, 52, 88, 106, 107, 122

[155] F. R. Vidal, N. Ivaki, and N. Laranjeiro. Analyzing the impact of elusive faults on blockchain reliability. *arXiv preprint arXiv:2304.05520*, 2023. → page 27

[156] J. Viega and G. R. McGraw. *Building secure software: How to avoid security problems the right way, portable documents*. Pearson Education, 2001. → page 3

[157] Vyper. Pythonic smart contract language for the evm. URL https://github.com/vyperlang/vyper. → pages 12, 34

[158] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu. Oracle-supported dynamic exploit generation for smart contracts. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1795–1809, 2020. → page 32

[159] X. Wang, H. Wu, W. Sun, and Y. Zhao. Towards generating cost-effective test-suite for ethereum smart contract. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 549–553. IEEE, 2019. → page 27

[160] Web. Governmental, 2016. URL https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck. → pages 6, 76

[161] Web. Decentralized application security project (or dasp) top 10, 2021. URL https://dasp.co. → pages 5, 6, 7, 78

[162] Web. Ethereum wiki: Ethereum contract security techniques and tips, 2021. URL https://eth.wiki/en/howto/smart-contract-safety. → page 81

[163] Web. contract-library, 2021. URL https://contract-library.com. → pages 91, 122

[164] Web. Dos with block gas limit, 2022. URL https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/#dos-with-block-gas-limit. → pages 15, 76, 81

[165] Web. Dos with failed call, 2022. URL https://swcregistry.io/docs/SWC-113. → pages 15, 77

[166] Web. Decentralized finance (DeFi), 2022. URL https://ethereum.org/en/defi. → page 2

[167] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt. Sok: Decentralized finance (defi). *arXiv preprint arXiv:2101.08778*, 2021. → page 2

[168] Why3. Why3. URL http://why3.lri.fr/. → page 33

[169] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014. → page 14

[170] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014. → pages 1, 2, 4, 13

[171] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen. Mutation testing for ethereum smart contract. *arXiv preprint arXiv:1908.03707*, 2019. → page 27

[172] V. Wüstholz and M. Christakis. Learning inputs in greybox fuzzing. *arXiv preprint arXiv:1807.07875*, 2018. → page 32

[173] V. Wüstholz and M. Christakis. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 789–800, 2020. → pages 32, 35

[174] J. Ye, M. Ma, Y. Lin, L. Ma, Y. Xue, and J. Zhao. Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. *Journal of Systems and Software*, 192:111410, 2022. → page 33

[175] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, pages 270–282, 2016. → page 34

[176] Z. Zhang, B. Zhang, W. Xu, and Z. Lin. Demystifying exploitable bugs in smart contracts. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, 2023. → pages 2, 7, 136

[177] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara. Security assurance for smart contract. In *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*, pages 1–5. IEEE, 2018. → page 35

[178] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 2019. doi:10.1109/TSE.2019.2942301. → pages 5, 15, 70