Towards Effective Static Analysis Approaches for Security Vulnerabilities in Smart Contracts

Asem Ghaleb
University of British Columbia, Canada
aghaleb@alumni.ubc.ca

ABSTRACT

The growth in the popularity of smart contracts has been accompanied by a rise in security attacks targeting smart contracts, which have led to financial losses of millions of dollars and erosion of trust. To enable developers discover vulnerabilities in smart contracts, several static analysis tools have been proposed. However, despite the numerous bug-finding tools, security vulnerabilities abound in smart contracts, and developers rely on finding vulnerabilities manually. Our goal in this dissertation study is to expand the space of security vulnerabilities detection by proposing effective static analysis approaches for smart contracts. We study the effectiveness of the existing static analysis tools and propose solutions for security vulnerabilities detection relying on analyzing the dependency of the contract code on user inputs that lead to security vulnerabilities. Our results of evaluating static analysis tools show that existing static tools for smart contracts have significant false-negatives and false-positives. Further, the results show that our first vulnerability detection approach achieves a significant improvement in the effectiveness of detecting vulnerabilities compared to the prior work.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*;

KEYWORDS

Ethereum, Solidity, smart contract vulnerabilities, bug injection, static analysis, taint analysis, data-flow analysis

ACM Reference Format:
Asem Ghaleb. 2022. Towards Effective Static Analysis Approaches for Security Vulnerabilities in Smart Contracts. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559567

1 INTRODUCTION

Smart contracts are programs written into blocks running on top of a blockchain that can receive and execute transactions autonomously. The growing interest in smart contracts is driven by their decentralized nature that enables secure distributed computations while eliminating the need for trusted third parties [23]. Smart contracts are written in Turing-complete programming languages, such as Solidity [13], and they are compiled to EVM low-level bytecode that is deployed to the blockchain. Each smart contract has access to a persistent storage to store data on the blockchain. Further, the execution of smart contracts on the Ethereum blockchain consumes gas paid for by users calling the contract functions.

As smart contracts are written by fallible human developers, like all software, they may contain bugs leading to vulnerabilities. However, the financial nature of smart contracts increases the importance of analyzing smart contracts for bugs. In real-world, Ethereum smart contracts manage accounts holding millions of dollars, and the most common domain where smart contracts are used is the decentralized finance (DeFi) [37]. Unfortunately, this financial nature provides a good incentive for malicious attackers to exploit vulnerabilities in smart contracts for financial gain [1, 3, 26]. The DAO vulnerability [1], for example, led to $60 million worth of Ether theft. Another vulnerability in the code of Parity's wallet was exploited by an attacker to steal $34 million worth of Ether [3].

Other than the financial losses characterizing vulnerabilities in smart contracts, the rules imposed by blockchain platforms, e.g., Ethereum, for controlling the update and execution of smart contracts increase the risk of vulnerabilities. For instance, transactions on smart contracts in Ethereum are immutable and cannot be reverted, so losses cannot be recovered. Further, it is difficult to update a smart contract after its deployment on the blockchain.

Several tools have been developed that statically find security vulnerabilities in smart contracts [9, 16, 22, 25, 27–29, 33, 34]. However, vulnerabilities abound in smart contracts [30], and studies [15, 17] show that existing tools are still in the infancy as they report high false alarms and fail to detect several vulnerabilities. Further, a recent study [38] found that developers rely on finding bugs in smart contracts manually, and it is time-consuming to manually identify bugs. Therefore, there is a compelling need for robust static analysis tools to help developers find security vulnerabilities in smart contracts before their deployment.

Having these issues in mind, the overarching goal of this PhD research is to *build robust and scalable static analysis approaches for detecting security vulnerabilities in smart contracts.*

2 PROBLEM DESCRIPTION AND OBJECTIVES

To achieve our goal, this research focuses on: (1) Understanding the current state of smart contract analysis tools; and (2) Proposing effective approaches for security vulnerabilities. In the following, we discuss the addressed research problem and our objectives.

2.1 Evaluating Smart Contract Static Analyzers

Despite the prevalence of smart contract static analysis tools, attack incidents against smart contracts exploiting code vulnerabilities

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559567

are reported frequently, and new exploits are still being discovered regularly [6]. This calls into question the efficacy of these tools and their associated techniques.

Understanding the efficacy of smart contract code analysis tools allows us to build effective techniques for detecting vulnerabilities in smart contracts. 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 vulnerabilities.

Objective: To address this problem, *the first objective of this thesis is to propose a systematic approach for evaluating smart contract static analysis tools based on bug injection.*

Related work: An empirical study by Durieux et al. [15] compared a number of smart contract static analysis tools. However, the study evaluated the tools using a dataset of example smart contracts covering limited cases of bugs rather than proposing a systematic evaluation approach. Bug injection as a testing approach has been explored considerably in the domain of traditional programs [8, 14, 31]; however, there have been limited work on bug injection in the context of smart contracts. In [7], Akca et al. used bug injection to evaluate the effectiveness of their proposed static analyzer. However, the approach is limited to injecting a single hard-coded bug snippet into a pre-defined location in the code. Injecting a single bug at specific location does not provide a comprehensive coverage evaluation of static analysis tools, and it does not lead to generating deep vulnerabilities and corner cases.

2.2 Detecting Security Vulnerabilities

We used our proposed systematic evaluation approach of smart contract static analysis tools to valuate the efficacy of a number of static analysis tools. By considering the results of our evaluation study and the research gaps, we focus in this research on the detection of two broad categories of vulnerabilities in smart contracts, namely, gas-related vulnerabilities and access control vulnerabilities.

2.2.1 Gas-Related Vulnerabilities. As mentioned above, the execution of a smart contract requires fees, called gas. The gas is paid by users upon transaction submission. Running out of gas during a transaction's execution results in exceptions and abortion of the transaction, thereby leading to unwanted behaviors. Further, Ethereum imposes an upper bound on the amount of gas spent to execute transactions (i.e., block gas limit). Therefore, when the block gas limit is exceeded during the execution of a transaction, the transaction fails, and its execution gets reverted, i.e., Ethereum rolls back all changes made before transaction failure. This prevents functions that exceed the gas block limit from completing successfully. The problem gets worse if these functions are responsible for transferring Ether out of the contract, as this would prevent the contract owners/users from ever accessing their money (Ether), e.g., in the Governmental contract [35], where about $2.5 million worth of Ether got locked out (freezed).

Smart contracts may contain code patterns that increase execution cost, causing the contracts to run out of gas or exceed block gas limit. These patterns can get used by malicious attackers to induce unwanted behavior in the victim contracts, e.g., Denial-of-Service (DoS) attacks [36]. We refer to these as *gas-related vulnerabilities*.

Discovering gas-related vulnerabilities in smart contracts by developers is not straightforward as the gas cost of a contract is based on (1) the cost of the bytecode low-level instructions rather than its source code and (2) the global state of the contract on the blockchain, i.e., data in the contract storage. There has been little prior work to find gas-related vulnerabilities in smart contracts [22]. However, these tools use pre-specified code templates and rules, and are hence brittle as even small variations in the patterns can make the tools fail, and also result in high false-positives.

Related work: MadMax [22], is the first tool to identify and detect gas-related vulnerabilities. It statically analyzes for gas-related vulnerabilities through the use of Datalog-based inference rules to search the intermediate representation (IR) of he bytecode for gas-related vulnerabilities. However, MadMax 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. Further, MadMax coarsely over-approximates many of the vulnerability cases, thereby resulting in high false-positives. Other tools, GasReducer [12], GASPER [11] and its extension, GasChecker [10], focus on optimizing gas consumption in smart contracts by detecting gas-inefficient code patterns and do not consider gas-related vulnerabilities.

Objective: Our second objective in this thesis is to *propose a detection approach that targets root causes of gas-related vulnerabilities without the need for any pre-existing code patterns and rules.*

2.2.2 Access Control Vulnerabilities. In smart contracts, developers rely on access control to manage who can call specific functions within the contract or execute critical instructions, e.g., withdraw money, kill the deployed contract. Smart contract developers implement access control checks in an ad-hoc manner employing different methods, such as using language-special-constructs (e.g., function modifiers in Solidity language) or conditional statements. Failing to implement needed access control properly results in several vulnerabilities in smart contracts due to weak and missing access control- we call these *access control vulnerabilities*. A known example is a hack that targeted Parity Wallet and led to locking out (freezing) about $280M worth of Ether [4]. The attack happened due to an access control vulnerability that enabled the attacker to reset the variables used to manage who is authorized to transfer money out of the contract.

Related work: Current tools that analyze smart contracts for access control vulnerabilities are limited to looking for specific predefined access control patterns in the code; hence ignore several vulnerabilities and report several false alarms. The work, Ethainter [9], performs information-flow analysis to detect five access control vulnerabilities; however, Ethainter searches for predefined access control patterns in the code, which results in several false-negatives and false-positives. Other tools [16, 27–29, 33, 34] partially target the problem and only check for the presence or lack of specific code patterns of access control before some pre-defined code statements. Further, these tools do not check for violations of the implemented access control. Finally, teEther [24] employs symbolic execution to generate exploits for a set of access-control-similar vulnerabilities; however, it scales to only a fraction of Ethereum smart contracts [9].

Objective: Our third objective in this thesis is to *build an efficient approach to infer the contract access control to detect access control vulnerabilities without relying on pre-defined specific code patterns.*

3 METHOD

This section describes how we address the discussed research problems and how we validate our proposed approaches.

3.1 Evaluating Smart Contract Static Analyzers

To achieve our first objective, we propose *SolidiFI* (Solidity Fault Injector) [17], a methodology for systematic evaluation of smart contract static analysis tools to discover potential flaws in the analysis tools that lead to undetected security bugs and false warnings. 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 bugs in contracts usually leads to loss of Ether (money). *SolidiFI* performs bug injection to evaluate the false-negatives and false-positives of smart contract static analysis tools. Our goal is to understand how effective are static analysis tools in detecting bugs in smart contracts. *To our knowledge,* SolidiFI *is the first systematic evaluation approach of smart contract static analysis tools.*

Bug injection in the context of smart contracts is a challenging problem 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 [13]. Second, because our goal is to inject security bugs in different potential locations, and the bugs injected should lead to exploitable vulnerabilities.

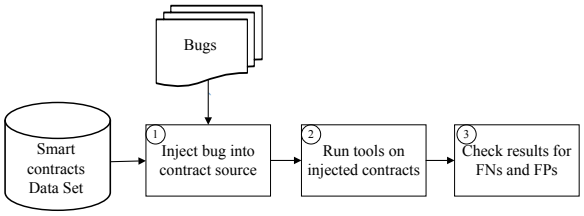

Figure 1: *SolidiFI* Workflow.

As shown in Figure 1, *SolidiFI* injects bugs formulated as code snippets into all possible locations in the smart contract's source code written in Solidity (step 1). The code snippets are vulnerable to specific security vulnerabilities that can be exploited by attackers. The resulting buggy smart contracts are then analyzed using the static analysis tools being evaluated (step 2), and the results are automatically inspected for those injected bugs that are not detected by each tool - these are the false-negatives of the tools - along with identifying the other bugs reported as false-positives (step 3). Our methodology is agnostic of the tool being evaluated.

The security bugs are injected into the source code in three different ways as follows. (1) *Full Code Snippet* approach in which we prepare several code snippets for each bug under study, where each code snippet is a piece of code that introduces the security bug. (2) *Code Transformation* approach that aims to transforming a piece of code without changing its functionality but makes it vulnerable to a specific bug. In this approach, we leverage known patterns of vulnerable code to inject this bug. (3) *Weakening Security Mechanisms* approach where we search for existing security mechanisms implemented in the code and weaken them to make the code vulnerable to specific security bugs.

3.1.1 Evaluation of SolidiFI. To evaluate our approach, we have selected six static analysis tools for evaluation, Oyente [25], Securify [34], SmartCheck [33], Mythril [28], Manticore [27], and Slither [16]. In our experiments, we injected bugs belonging to seven different bug types that have been exploited in practice [1, 2, 5], and they are within the detection scope of the selected tools. To perform our experiments, we used a data set of fifty real-world smart contracts deployed on Ethereum blockchain.

False-negatives and false-positives of the evaluated tools. The results of injecting bugs of each bug type and testing them using the six tools show 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 bugs correctly.* Further, results show that *all the evaluated tools have reported several false-positives for most of the bug types.* Interestingly, the results show that the tools with low numbers of false-negatives reported high false-positives. This raises the question of whether the high detection rate was simply due to over-zealously reporting bugs by these tools.

To establish a practical understanding of why set of bugs were not detected, we studied the code snippets for some of the undetected bugs. We found that using pattern matching for detecting bugs is not an effective way for detecting all smart contract bugs as several bugs cannot be expressed as specific patterns. Further, 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 other aspects of the analyzed code instead of depending only on analyzing the syntax and symbolic traces. – For more details, please reference SolidiFI's paper [17].

3.2 Detecting Gas-Related Vulnerabilities

For our second objective, we propose *eTainter* (EVM Tainter) [18], an efficient static analysis-based approach that uses taint tracking to find gas-related vulnerabilities without using pre-specified rules. The novel key insight in our work is that the common root cause of the gas-related vulnerabilities is the gas exceptions triggered due to the dependency of the contract code on data items either provided or manipulated by the contract users. We, therefore, formulate the detection of gas-related vulnerabilities as a taint analysis problem [32] and use static *taint tracking* to find gas-based vulnerabilities, without any pre-existing code patterns and rules [18]. Thus, we first define the critical statements in the contract code in which gas exceptions can cause any of the gas-related vulnerabilities. Then, we use static taint analysis to track the dependency of these critical statements on user data inputs that cause gas exceptions. *To the best of our knowledge,* eTainter *is the first technique to target gas-related vulnerabilities without requiring pre-specified code patterns via static taint analysis.*

At a high-level, *eTainter* takes as input the bytecode of the smart contract being analyzed; it builds the control flow graph (CFG) and identifies the instructions that introduce user data in the bytecode (taint sources) and the instructions that can form gas-related vulnerabilities (sinks). It then extracts the CFG paths leading to the defined sinks and performs taint analysis on these paths. Finally,

eTainter reports the found vulnerabilities to the users, along with the corresponding vulnerable functions causing the vulnerabilities.

Unlike traditional programs, performing static taint analysis in smart contracts, particularly in the contract bytecode, is faced with several challenges. Smart contracts have certain peculiarities (e.g., taint propagation via contract's persistent storage, multiple entry points, access control) and patterns of use (e.g., use of cryptographic hash to to reference arrays and mappings stored in the contract storage). *eTainter* addresses these challenges to avoid false-positives and false-negatives – details are discussed in eTainter's paper [18].

3.2.1 Evaluation of eTainter. We evaluate *eTainter* first by comparing it with the prior work, MadMax, in terms of its effectiveness in detecting gas-based vulnerabilities on a dataset having ground truth of the existing vulnerabilities in the contracts. We use the reported results to estimate the precision, recall, and F1 score (harmonic mean of the precision and recall) for both tools.

Second, we use *eTainter* to perform a large-scale analysis of 60,612 real-world contracts on the Ethereum blockchain to measure the performance of *eTainter* and to determine how prevalent are gas-related vulnerabilities in real-world contracts. We also run *eTainter* on the 3000 most popular contracts on Ethereum to determine how prevalent are gas-related vulnerabilities in widely-used contracts.

Comparison with the prior work. The results of comparing our approach with the prior work, MadMax, show that eTainter *reported more true-vulnerabilities and less false alarms compared to MadMax.* Overall, *eTainter* has a precision of 90.4%, a recall of 94%, and an F1 score of 92.2%. In comparison, MadMax has a precision of 64.9% and a recall of 74%, which leads to an F1 score of 69.2%.

Performance of *eTainter*. The analysis results of the 60,612 contracts show that *eTainter* has an average analysis time of 8 seconds and a memory consumption of 118MB per contract.

Prevalence of gas-related vulnerabilities. *The results indicate that the addressed vulnerabilities are prevalent in real-world contracts.* *eTainter* flagged more than 2,834 real-world contracts as having gas-related vulnerabilities. These contracts are vulnerable to DoS attacks that can result in blocking the use of the contracts forever.

3.3 Detecting Access Control Vulnerabilities

The main challenges for effectively detecting access control vulnerabilities in smart contracts are to (1) accurately identify various forms of access control checks; and (2) precisely distinguish the vulnerable access control checks from intended behavior cases – when unauthorized users can manipulate access control data as intended behavior, e.g., buying the contract ownership.

To address these challenges, we are working on building an efficient approach for detecting access control vulnerabilities that is (1) independent of the various code patterns used to implement access control checks; and (2) able to differentiate access control vulnerabilities from intended behavior cases. Our key insight for identifying access control checks is that access control checks can get inferred from the semantics of the instructions contributing to the evaluation of the conditions forming access control checks and from the data dependencies between the state variables storing access control data (used in the conditions) and the contract functions. Thus, analyzing data flows of conditions in a smart contract can be used to determine access control checks in the smart contract. That is, we first use static data-flow analysis to accurately define access control checks without relying on pre-defined access control patterns. Then we use static taint analysis to check if unauthorized users can control the identified access control checks.

To reduce false-positives, we analyze code for cases that are part of the contract functionality and separate them from vulnerabilities. We use symbolic execution-based analysis to infer cases where the contract implements non-access control constraints to control manipulating access control data and consider them as benign cases intentionally implemented rather than vulnerabilities.

3.3.1 Evaluation plan. Our initial plan is to evaluate the effectiveness and performance of the proposed approach by answering the following research questions:

RQ1. What is the effectiveness of the proposed approach compared to prior work?

RQ2. What is the performance of the proposed approach?

RQ3. How efficient is the approach to find real vulnerabilities?

To compare the proposed approach with the prior work, we will use a dataset with a ground truth of the vulnerabilities. Further, we will study the performance of the approach and how useful it is to find true vulnerabilities by performing large-scale analysis on the 10,000 most-used contracts deployed on the Ethereum blockchain.

4 EXPECTED CONTRIBUTIONS

The expected main contributions of this research are as follows.

- A systematic evaluation methodology of smart contract static analysis tools.
- An inter-procedural static taint-analysis approach for smart contract bytecode that considers smart contract special concepts such as tracking taints through the contract's persistent storage in addition to using domain-specific optimizations to reduce false-positives.
- A data-flow analysis-based approach for inferring the access control checks implemented in smart contracts without relying on pre-defined code patterns.
- Three automated tools [19, 21], built in this research, and several benchmarks [20, 21] that are publicly available to be used by future studies.

5 CONCLUSION AND FUTURE RESEARCH

This thesis aims to help the development of secure smart contracts. We propose a methodology for systematically evaluating static analysis tools based on bug injection. Further, we propose two static analysis approaches for detecting gas-related vulnerabilities and access control vulnerabilities, based on studying the dependency of the contract code on user inputs causing these vulnerabilities.

As future work, we plan to expand our work to involve cross-contract (inter-contract) static analysis approaches. This will help improve the analysis effectiveness of contracts relying on the code of other smart contracts, e.g., for code reusability.

ACKNOWLEDGMENTS

We thank Karthik Pattabiraman and Julia Rubin for their guidance and support during this work and the anonymous reviewers of the ASE'22 doctoral symposium for their helpful comments.

REFERENCES

[1] 2016. The DAO smart contract. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[2] 2017. History of Ethereum Security Vulnerabilities, Hacks, and Their Fixes. https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes

[3] 2017. The parity wallet breach. https://bitcoinexchangeguide.com/parity-wallet-breach.

[4] 2018. Accidental's bug froze $280 million worth of ether in Parity wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[5] 2018. New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018–10299). https://medium.com/@peckshield/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536

[6] 2022. Smart Contract Weakness Classification. https://swcregistry.io.

[7] Sefa Akca, Ajitha Rajan, and Chao Peng. 2019. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 482–489.

[8] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering flaws in security-focused static analysis tools for Android using systematic mutation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1263–1280.

[9] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[10] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2020. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing* (2020).

[11] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[12] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. 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)*. IEEE, 81–84.

[13] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Vol. 1. Springer.

[14] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 110–121.

[15] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[17] Asem Ghaleb and Karthik Pattabiraman. 2020. 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*. 415–427.

[18] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: Detecting Gas-Related Vulnerabilities in Smart Contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 728–739.

[19] GitHub. 2020. SolidiFI. https://github.com/DependableSystemsLab/SolidiFI

[20] GitHub. 2020. SolidiFI Benchmark. https://github.com/DependableSystemsLab/SolidiFI-benchmark

[21] GitHub. 2022. eTainter. https://github.com/DependableSystemsLab/eTainter

[22] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[23] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[24] Johannes Krupp and Christian Rossow. 2018. {teEther}: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1317–1333.

[25] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[26] Florian Mathieu and Ryno Mathee. 2017. Blocktix: decentralized event hosting and ticket distribution network. https://www.cryptoground.com/storage/files/1527588859-blocktix-wp-draft.pdf.

[27] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[28] Bernhard Mueller. 2022. Mythril. https://github.com/ConsenSys/mythril. https://github.com/ConsenSys/mythril

[29] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*. 653–663.

[30] Daniel Perez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? *arXiv preprint arXiv:1902.06710* (2019).

[31] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 214–225.

[32] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[33] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[34] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[35] Web. 2017. GovernMental. https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck

[36] Web. 2022. Decentralized Application Security Project (or DASP) Top 10. https://dasp.co

[37] Web. 2022. Decentralized finance (DeFi). https://ethereum.org/en/defi

[38] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* (2019).