Fuzzing for Smart Contract Interworking Security Evaluation: An Empirical Evaluation of the State of the Art

Simone Zerbini
simone.zerbini@unipd.it
Università degli Studi di Padova
Padova, Italy

Sahil Kharya
skharya@asu.edu
Arizona State University
Tempe, Arizona, USA

Dragan Boscovic
dragan.boscovic@asu.edu
Arizona State University
Tempe, Arizona, USA

Eleonora Losiouk
eleonora.losiouk@unipd.it
Università degli Studi di Padova
Padova, Italy

ABSTRACT

The smart contract development provides various features and solutions to the end user including transparency, immutability and many more. They play a pivotal role in building various decentralized applications such as DeFi platforms and have shown a tremendous growth with a large sum of money being circulated. With the vast adoption, various vulnerabilities arises, causing serious concerns with the security and resulting in huge financial loss. To mitigate these issues, several researches have been conducted and different testing methodologies are utilized to find and resolve the vulnerabilities. In this paper, we look into one of the testing method called fuzzing for the discovery of vulnerabilities in smart contracts. Our research embarks on a comprehensive investigation of modern fuzzing tools, emphasizing their capability to analyze interacting contracts and handle complex contract deployment steps. Both these features are fundamental for properly analyzing large multi-contract projects, which are becoming increasingly common in the web 3.0 technologies scenario. This research highlights the tools' effectiveness, and ability to analyze large multi-contract projects, shedding light on their limitations.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

Smart Contracts, Fuzzing, Blockchain

ACM Reference Format:
Simone Zerbini, Sahil Kharya, Dragan Boscovic, and Eleonora Losiouk. 2023. Fuzzing for Smart Contract Interworking Security Evaluation: An Empirical Evaluation of the State of the Art. In *Proceedings of WEB3SEC Workshop Encouraging Building Better Blockchain Security (WEB3SEC'23).* ACM, Austin, Texas, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

WEB3SEC'23, December 2023, Texas

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Smart contracts are self-executing contracts in which the conditions or terms of the agreement are expressed as lines of code. These contracts are executed in a transparent, unreachable, and decentralized manner through the use of blockchain platforms like Ethereum. Smart contracts are essential in the field of blockchain technology for enabling safe and decentralized transactions. However, thorough testing techniques are required due to the complexity of smart contracts and the possibility of security flaws. With the growing adoption of Web 3.0 technologies, ensuring their security and correctness is paramount. Fuzzing is one technique that is gaining popularity for finding possible flaws in smart contracts by subjecting them to a barrage of random inputs. It's a quick, effective, and brute-force method of testing edge cases and scenarios. Fuzzing presents several challenges such as code coverage or the creation of meaningful inputs; in addition, the state-based nature of smart contracts introduces new challenges. In recent years, many new fuzzing tools for smart contracts have been proposed with the intention of solving these challenges [14, 31, 25, 19, 4, 27, 34, 16]. However, these works often focus heavily on the performance of the proposed new tool, such as code coverage an detection rate, neglecting other very important aspects such as the level of human intervention required by the tool or the ability to analyze large multi-contract projects.

In this paper, we present an overview of the vulnerabilities that can affect a smart contract and the most relevant fuzzing tools currently available (including some very recent ones) [14, 31, 25, 19, 4, 27, 34, 16]. Afterwards, an empirical evaluation of 5 of these fuzzing tools was conducted, namely Echidna [14], ItyFuzz [27], ConFuzzius [31], EF/CF [25] and Smartian [4]. Unfortunately, it was not possible to perform this evaluation on every tool since some of them were not fully available. In particular, this evaluation was focused on the capability to analyze interacting smart contracts and to handle projects that require complex steps for deployment. Both these features are fundamental for properly analyzing large multi-contract projects.

The capability to properly model and analyze interacting smart contracts is critical because there might exist some contract states that can only be reached by interacting with a second contract.

On top of that, a contract could present a vulnerability only in one of these states. In this scenario, if the fuzzing tool fails to model properly the interaction between contracts, it will not be able to reach the vulnerable state, i.e. it will not be able to detect the vulnerability. Contract state refers to the values of the contract's persistent variables, which affect the behavior of the contract itself. The empirical evaluation of this aspect was carried out through 6 test cases designed to highlight the limitations of the tools in analyzing interacting smart contracts. Each test case contains an artificial vulnerability, which can be triggered only by a specific sequence of transactions involving multiple contracts. It should be noted that our test cases all contain the same artificial vulnerability, because we are not interested in the vulnerability itself but in the way it is triggered.

The ability to handle complex deployment steps is vital since contracts often require some specific operations to be deployed (such as constructor parameters, function calls, storing addresses of other contracts) and sometimes (especially in a large project) they can become very complex. These operations affect the internal state of the contracts and must therefore be taken into account during fuzzing analysis. For this reason, we looked for features that allow the contracts under analysis to be set to a specific initial state. We based the evaluation of this aspect on our observations gathered during the previous analysis and on the information collected from papers and documentations.

The rest of the paper is structured as follows. In Section 2 we discuss the related work. Section 3 gives some knowledge about smart contracts and fuzzing. Section 4 gives an overview about smart contracts vulnerabilities. The fuzzing tools are described briefly in Section 5. Section 6 presents our methodology and test cases. In Section 7 we discuss our results. Finally, Section 8 concludes the paper.

2 RELATED WORK

Smart contracts validation is a very active research field. Many fuzzing tools have been proposed in recent years [14, 31, 25, 19, 4, 27, 34, 16]. Beyond fuzzing, other techniques have been widely exploited, such as symbolic execution [20, 32, 21, 23, 24, 22] and static analysis [12, 3]. These papers, however, only compare the performance of the different solutions, ignoring other aspects that are very important, especially in a production environment. In Durieux et al. [8] a survey of several validation tools followed by an empirical evaluation is presented, but even this survey only evaluates performance.

In recent years, many survey papers focusing on various aspects of smart contract security have been published. These include reviews of the potential smart contracts vulnerabilities [2, 29] and the tools used to test their security [1, 5, 8, 30, 26, 15]. But most of these surveys simply conduct a review of the literature.

In this work, we aim to test important features such as the level of automation, and the capability to analyze large multi-contract projects. For the moment, we have limited ourselves to evaluating only fuzzing tools, but we aim to extend the work to other techniques as well.

3 BACKGROUND

3.1 Smart Contracts

Smart contracts are the programs which are used for developing decentralized applications (DApps) on top of blockchain's consensus protocols. These contracts facilitate users to craft and uphold agreements on the blockchain, substantially diminishing the need for trust in third-party intermediaries. Among various blockchain layers available, Ethereum is the most popular and commonly used blockchain. It is layer-1 chain and works on the Ethereum Virtual Machine(EVM). To utilise the features of EVM, smart contracts are developed using a dedicated high-level programming language, such as Solidity, that compiles into low-level bytecode. After developing a smart contract with the required definitions, it is deployed. The contract definitions are final and cannot be removed or updated after deployment, making smart contracts essential in facilitating secure and decentralized transactions.

Utilizing the smart contracts, users can develop decentralized applications(DApps). DApps are complex systems that include both frontend and backend elements. At its heart, smart contracts serve as the application's backbone, defining its logic and procedures. On the other hand, the user interfaces, which are designed with end users in mind, make up the frontend and make the DApp's services accessible and usable.

3.2 Fuzzing

Fuzzing is an automated testing technique that involves generating a large number of semi-random inputs and then submitting them to a system to identify vulnerabilities. It is a fast and efficient way to test edge cases and scenarios that might not be considered during manual testing.

Unfortunately, fuzzing presents some unsolved challenges, such as code coverage. The term code coverage refers to the portion of the code of the software under analysis that has been reached/tested by the fuzzing tool. Often software contains pieces of code that are triggered only if a specific condition is met. The problem is that, if the fuzzing tool cannot pass these conditions, a large portion of the code will not be tested, thus compromising the efficacy of the analysis. Another common challenge is the creation of meaningful inputs. In fact, whenever the (pseudo)random generation of a new input fails in creating a valid one, the analysis will make no progress, thus wasting time.

Moreover, the smart contracts scenario presents different challenges. Although the amount of source code of most smart contracts is trivial compared to complex software like browsers and operating systems, they are stateful and have complex dependencies with other smart contracts [27]. In fact, the behavior of smart contracts depends on their internal state and the state of the blockchain. These states, especially the contract state, change depending on the order and parameters of called functions. This means that it is no sufficient for the fuzzing tool to generate a single valid input but, instead, it is required a sequence of valid inputs. Furthermore, this also implies that, in order to comprehensive analyze a smart contract, it is not sufficient to extensively test all functions but, instead, it is required to extensively test all functions with all possible states that the contract can assume.

4 SMART CONTRACTS VULNERABILITIES

In this section, we give an overview of the vulnerabilities that can affect a smart contract. It should be noted that these are vulnerabilities of universal relevance and that a smart contract might present other bugs or vulnerabilities specific to its application scenario.

4.1 Integer Overflow and Underflow

Integer overflows or underflows occur, and the result becomes an unexpected value. In an overflow, a value becomes too large and wraps around, resetting at a low number. In underflow, it is the opposite: a value goes below what is allowable and wraps to a high number. It often occurs when a developer disregards the variable's bounding value, leading to a value that exceeds either the upper or lower bound of the variable. This vulnerability can help an attacker to transfer many tokens at a cheap rate. The SafeMath library in Solidity should be used to prevent the issue of integer overflow and underflow.

4.2 Reentrancy

A function in a victim contract is re-entered and leads to a race condition on state variables. It is a technique for exploiting smart contracts that enables an attacker to repeatedly call a contract function (similar to recursive call of a function), creating an endless loop, until the gas is exhausted or the balance depletes completely and perhaps stealing money. Reentrancy was exploited in the DAO attack, i.e. the biggest attack on Ethereum smart contracts [33].

4.3 Transaction order dependency

Transaction order dependency means carrying out the transactions in a defined sequence so that to get a required result [20]. Inconsistent transactions execution with respect to the time of invocations may deliver different results. Usually, several transactions are collected together into blocks on blockchains, and the transactions inside a block are not necessarily handled in the order they were received. This is because miners can select randomly a transaction from the block based on the high gas price. Benefiting from this flaw, an attacker attacks by introducing a transaction with high gas cost based on the visible pending transactions in order to gain control of a known outcome.

4.4 Block State Dependency

Block states (e.g. timestamp, number) are often used in smart contracts for various purposes, such as determining the time elapsed between events or generating random numbers. It can also decide ether transfer of a contract. (or modify the behavior of an instruction create, call, delegatecall, or selfdestruct). A miner might possibly change the timestamp by several seconds, and a contract's process may be predictable if it relies entirely on block properties. These conditions make the smart contract vulnerable. By changing the block timestamps directly, malicious actors (miners) can take advantage of the vulnerability when a smart contract transfers ETH using block.timestamp, creating the ideal environment for their own success [19]. Whenever possible, avoid using block.timestamp or block.number for critical decision-making processes. Another way would be to use a reliable algorithm for the production of random numbers.

4.5 Exception Disorder

A contract does not check for an exception when calling external functions or sending ether [20]. The vulnerability is caused by calls between smart contracts using functions such as <address>.send, <address>.call.value, or calling methods of other contracts using statements such as call. Running out of gas, for example, might cause an exception to the call, in which case the call would end and the state will roll back, returning false to the calling contract. To mitigate, its suggested to use revert to revert all changes in case of failure. Refrain from using low-level calls, one can use.transfer() rather than.call(), which produces an exception upon failure and halts execution.

4.6 Control-Flow Hijack

An attacker can arbitrarily control the destination of a jump or delegatecall instruction [19]. The delegatecall function is a powerful feature in Solidity, allowing one contract to execute code in the context of another contract. Delegate calls are critical in terms of smart contract security because the logic in the callee contract might change the storage of the caller contract. While using delegatecall, ensure the state variables are declared in the same order to prevent unnecessary updates to other state variables which can provide the base for the attacker.

4.7 Leaking Ether

A contract allows an arbitrary user to freely retrieve ether from the contract [23]. It happens because of a flawed logic or insufficient access constraints that let unauthorized individuals to freely withdraw the ether from the contract without having owner rights. It results in financial loss and operational disruption as well as makes a dent in user's trust. Few mitigation strategies include using modifiers to restrict access, limiting exposure to ether with withdrawal limit, and most important auditing and testing for the smart contract.

4.8 Locking Ether

A contract can receive ether but does not have any means to send it out [23]. Ether will become locked in the contract indefinitely if the contract receiving the ether provides a fallback mechanism that prevents ether from being withdrawn or a function that locks the ether without enabling it to be withdrawn. In this case, ether becomes unintentionally trapped in a smart contract, rendering it inaccessible and essentially removing it from circulation. This has resulted in a huge loss for various users.

4.9 Unprotected self-destruct

An arbitrary user can destroy a victim contract by running a selfdestruct instruction [23]. Since, the contracts once deployed, it cannot be updated and will continue to to function as defined before deployment, Solidity provides a selfdestruct() function which allows the owner to destroy the contract if something went wrong. However, if a smart contract with an unprotected self-destruct function lacks proper permission control, any user can potentially exploit this vulnerability to call the self-destruct function and transfer the contract's Ether to their desired address, leading to funds loss. Thus, the proper permission controls should be implemented for important functions like selfdestruct.

4.10 Arbitrary Write

An attacker can overwrite arbitrary storage data by accessing a mismanaged array object. This can lead to overwriting of adjacent storage slots that might hold other critical state variables. This may disrupt the contract's intended logic and result in monetary losses or other unforeseen consequences. We must make sure that the given index is within the boundaries of the array in order to reduce this risk. In addition, the use of modifiers can limit access or verify conditions before a function is called.

4.11 Multiple Send

A contract sends out ether multiple times within one transaction, especially in loops. This is a specific case of DoS. It can lead to various problems like if the transaction exceeds the block's gas limit, then it will fail. Also, a malicious actor could exploit all the calls in a loop by making high gas costs leading to a failure of the complete transaction. This vulnerability may also cause the contract to be in an inconsistent state.

4.12 Transaction Origin Use

A contract relies on the origin of a transaction (i.e. tx.origin) for user authorization. The problem is that tx.origin does not always correspond to the direct caller of a function, this can lead to a form of phishing attack that can drain a contract of all funds. To prevent this vulnerability, it is advisable to use msg.sender instead of the tx.origin.

5 TOOLS REVIEW

In this section we briefly describe some of the most relevant existing fuzzing tools. We have evaluated only Echidna, ItyFuzz, ConFuzzius, EF/CF, and Smartian but, for the sake of completeness, we also provide descriptions of other relevant tools.

5.1 Echidna

Echidna [14, 9] is an open-source Ethereum smart contract fuzzer published by Trail of Bits. Echidna leverages Slither [12], a smart contract static analysis framework, to compile the contracts and analyze them to identify useful constants and functions that handle Ether directly. After this pre-processing step, the fuzzing campaign starts. Rather than relying on a fixed set of pre-defined bug oracles to detect vulnerabilities, Echidna supports three types of properties: user-defined invariants, assertion checking, and gas use estimation. Currently, Echidna can test both Solidity and Vyper smart contracts.

One of the main goals in developing of Echidna, was to make it easy to use, configure and integrate into the contract development workflow. This makes it one of the best options in a production environment. In addition, several tools are supplied by Echidna in order to deal with complicated contracts, for example Etheno [11]. Etheno is a tool aimed at removing the complexity of setting up Echidna on large multi-contract projects. It can be used to record the sequence of operations required to deploy the DApp so that Echidna has a good starting point.

5.2 ItyFuzz

ItyFuzz [27, 18] is a snapshot-based fuzzing tool. Snapshots are essentially replicas of intermediate states of the contracts. By storing all interesting snapshots, ItyFuzz can "time travel" to previous states without re-executing the operations required to build them up. Time traveling allows for efficient exploration for search space of both transactions and states. ItyFuzz offers different ways to handle complex deployment steps, such as the option to assign a specific address to each contract, or the ability to take a snapshot of a state pulled from a specific block from any blockchains supporting Geth client. In addition, ItyFuzz provides both an exhaustive bug oracle and the support for user-defined invariants.

5.3 ConFuzzius

ConFuzzius [31, 6] is an hybrid fuzzer which combines fuzzing and symbolic execution. Symbolic taint analysis is employed to generate path constraints on tainted inputs. The moment the fuzzer stops progressing, a constraint solver is deployed to solve the constraint in question. These solutions are collected within a mutation pool, from which the fuzzer can draw to move past the challenging contract condition.

ConFuzzius models the return values of calls to other contracts as fuzz-able inputs. The same thing is done for the blockchain state, which contains values such as block number and block timestamp. Moreover, it implements an exhaustive bug oracle but it does not support user-defined invariants.

5.4 EF/CF

Extremely Fast Contract Fuzzer (EF/CF) [25, 10] is not only a fuzzing framework but also an exploit generator. EF/CF holds true to its name; in fact, it can reach in seconds sections of code that the other tools take hours to reach. In contrast to other fuzzing tools that rely on heuristics to detect reentrancy vulnerabilities, EFCF simulates the behavior of multiple attacker-controlled smart contracts. In this way, the response of EF/CF will be much more accurate about the actions needed to trigger the vulnerability. In order to achieve better code coverage and make the tool scale even on more complex contracts, EF/CF bets on speed, that is, on generating new inputs as fast as possible. To increase the fuzzing throughput, EF/CF translates the Ethereum virtual machine (EVM) bytecode to equivalent C++ code and uses a high-performance coverage-guided fuzzer (AFL++ [13]). EF/CF utilizes a simple yet powerful bug oracle based on Ether gains. In addition to detecting unprotected self-destruct and control-flow hijack vulnerabilities, this bug oracle monitors the ether balance of the attacker-controlled contracts. If the sum of the balances exceeds the initial value, then an attack has occurred. However, this oracle do not cover all current smart contract security issues. In order to extend EF/CF to cover smart contract specific bugs, EF/CF allows developers to define custom bug oracles in their Solidity code.

Similarly to ConFuzzius, EF/CF models the state of the blockchain and the values returned by other contracts as fuzz-able inputs.

5.5 Smartian

Smartian [4, 28] is a fuzzing tool that leverages static analysis to identify significant transaction sequences by the data dependencies between functions and persistent state variables. The idea is that a transaction that modifies the state variables of a contract is more interesting because it can help reach deeper contract states. Smartian provides an exhaustive bug oracle but it does not support user-defined invariants. Moreover, it cannot analyze interacting smart contracts nor handle complex deployment steps.

5.6 Harvey

Harvey [34] is an industrial fuzzer developed by ConsenSys. It is used at ConsenSys both for smart-contract audits and as part of MythX, an automated contract analysis service. It has analyzed more than 3.3M submitted contracts from March 2019 to April 2020 and has found hundreds of thousands of issues. Unfortunately, Harvey is not open source.

5.7 ILF

ILF [16, 17] is a fuzzing tool based on the imitation learning concept. More specifically, in order to generate good inputs, the fuzzer leverages a neural network trained on a dataset of inputs obtained by running a symbolic execution expert on tens of thousands of contracts. In this way, the fuzzer can combine strengths of both fuzzing and symbolic execution - it generates effective inputs quickly. Unfortunately, the training set is not provided along with the tool.

ILF supports fuzzing multiple interacting contracts that are deployed on the local test blockchain. It implements a bug oracle which, even if it does not cover all vulnerabilities, can be expanded.

5.8 ContractFuzzer

ContractFuzzer [19, 7] is the first tool applying fuzzing techniques to smart contracts for vulnerability detection. ContractFuzzer does not provide the support for user-defined invariants, but only a quite limited bug oracle. Moreover, it cannot analyze interacting smart contracts nor handle complex deployment steps.

6 METHODOLOGY

Our goal was to evaluate fuzzing tools on 2 aspect: the capability to properly analyze interacting contracts, and the capability to handle project with complex deployment steps. For the first aspect, we based our evaluation on 6 test cases. These test cases are designed to stress the tools and highlight their limitations, but at the same time they represent very trivial and common contract interaction patterns. In fact, most test cases simply consist of a contract that switches behavior based on the value returned by a function call to another contract. Although these test cases are very simple (1 or 2 functions each), we will see that they are enough to pose a challenge to most fuzzing tools. In each test case we placed a vulnerability, in particular, we chose to use unprotected self-destruct. This choice was made because we needed a vulnerability that could be easily detected by all tools and that we could easily insert at our convenience. The type of vulnerability adopted is not relevant since we are more interested in the sequence of operations needed to trigger it than in the detection of the vulnerability itself.

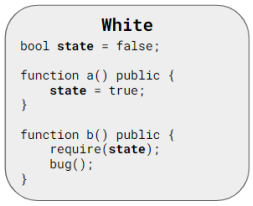

Figure 1: Diagram representing the White test case. Transactions sequence for vulnerability: White.a(), White.b().

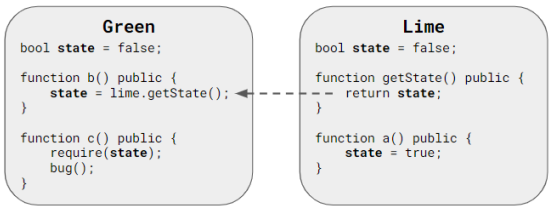

Figure 2: Diagram representing the Green test case. Transactions sequence for vulnerability: Lime.a(), Green.b(), Green.c().

For the second aspect, that is the capability to handle project with complex deployment steps, we looked for features that allow the contracts under analysis to be set to a specific initial state. We based the evaluation of this aspect on our observations gathered during the previous evaluation and on the information collected from papers and documentations.

6.1 Test case White

This first test case consists of a simple contract in which we placed the vulnerability, as shown in Figure 1. The White test case was used mainly to verify that the tools were able to detect the vulnerability we chose and that they were set up properly.

6.2 Test case Green

This test case, as shown in Figure 2, consists of 2 interacting contracts: Green and Lime. The Green contract contains the vulnerability, but it can only be triggered if a certain function of the Lime contract is invoked first. The addresses of the contracts are hardcoded into the constructor. In this case, the vulnerability can be detected even if the fuzzing tool does not execute the code of the second contract (Lime), but generates random return values. This is why the following Black test case was created.

6.3 Test case Black

This test case, as shown in Figure 3, consists of 2 interacting contracts: Black and Gray. The Black contract contains a vulnerability that can only be triggered if the Gray contract returns True. The Gray contract, however, will always return False. So, actually, there is no vulnerability. Contract addresses are hardcoded into the constructor. In this case, if the fuzzing tool does not execute the code of

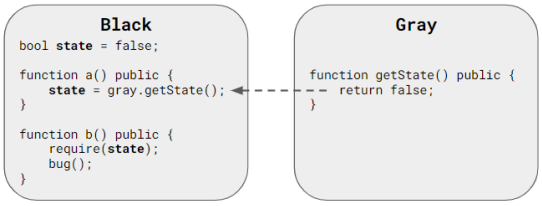

Figure 3: Diagram representing the Black test case. The vulnerability cannot be triggered.

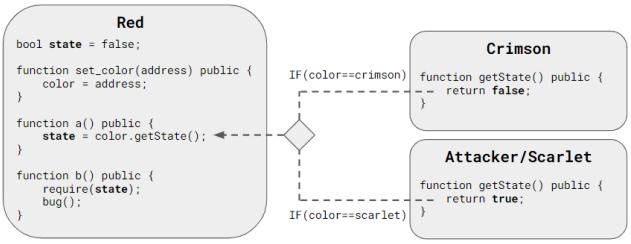

Figure 4: Diagram representing the Red and Scarlet test cases. Transactions sequence for vulnerability: Red.set_color(attacker or scarlet address), Red.a(), Red.b().

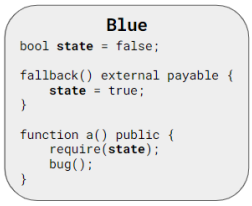

Figure 5: Diagram representing the Blue test case. Transactions sequence for vulnerability: Blue.fallback(), Blue.a().

Tool	White	Green	Black	Red	Scarlet	Blue
Echidna	1	1	1	0	1	1
ItyFuzz	1	1	1	1	1	1
ConFuzzius	1	x	0	0	0	1
EF/CF	1	x	0	x	x	1
Smartian	1	0	0	0	0	1

Table 1: This table shows the result obtained with our empirical evaluation: 1 = Success, 0 = Fail, x = bug detected but incorrect tx sequence.

the second contract (Gray) but simply returns random values, the vulnerability will be detected. It will, however, be a false positive.

6.4 Test case Red

This test case, as shown in Figure 4, consists of 2 interacting contracts: Red and Crimson. The Red contract contains a vulnerability that can only be triggered if the Crimson contract returns True. The Crimson contract, however, will always return False. This test case is similar to the previous one, but in this case the address of the Crimson contract is passed to the Red contract via a public method. This opens up the possibility for an attacker to create a third contract that presents the same interface of the Crimson contract but returns True instead.

6.5 Test case Scarlet

The Scarlet test case is almost the same as the previous one, but this time there is also the code of a third contract (Scarlet) with the same interface as the Crimson contract, but which returns True.

6.6 Test case Blue

In the Blue test case, as shown in Figure 5, the vulnerability is hidden behind a fallback function. We implemented 2 contracts: Blue and Cyan, with the vulnerability triggered by the `receive` and `fallback` functions respectively.

7 RESULTS

The results obtained from the analysis on our test cases are reported in Table 1. It is evident from the table that Echidna and ItyFuzz are the only tools capable of analyzing multiple interacting contracts. Moreover, it can be noted that Echidna fails in analyzing test case Red, while ItyFuzz succeeds. This indicates that Echidna fails to theorise the existence of other contracts outside of those provided during the analysis, this capability makes ItyFuzz's analysis much more powerful.

Still regarding test case Red, we can see that EF/CF manages to detect the bug, even though it provides an incorrect transaction sequence. This is due to the use of an approximation in the interaction between contracts. This approximation causes incorrect transaction sequences (as in Green, Red and Scarlet) and several false positives (as in the case of Black), but, in test case Red, it succeeds in alerting the developer where Echidna fails.

In Table 2, a summary of the features provided by each tool is given. Bug Oracle refers to the capability to detect known bugs of universal validity, such as those described in section 4. It can be seen that Echidna is the only tool that lacks a bug oracle. This is a major shortcoming of Echidna, since it assumes that developers are aware and updated on all vulnerabilities and are able to define rules that cover all of them.

The Invariants column refers to the capability to define custom rules that must always be respected during execution. This allows developers to verify that the contract never reaches states that are considered faulty. This is a critical feature since a generic bug oracle may not be sufficient to detect errors specific to the contract.

By Complex Deploy we refer to the capability to analyze DApp consisting of multiple smart contracts, and thus requiring complex deployment and set up operations. Echidna provides this feature with the help of Etheno, while ItyFuzz thanks to its snapshot-based architecture.

Finally, the last column summarizes the results obtained through our evaluation regarding the capability to analyze multiple interacting contracts.

This table shows that Echidna and ItyFuzz are the most comprehensive tools, with only ItyFuzz providing all the features.

Tool	Bug Oracle	Invariants	Complex Deploy	Inter-Contract
Echidna	No	Yes	Yes	Yes
ItyFuzz	Yes	Yes	Yes	Yes
ConFuzzius	Yes	No	No	No
EF/CF	Yes	Yes	No	No
Smartian	Yes	No	No	No

Table 2: Summary of our tools evaluation.

8 CONCLUSIONS

In conclusion, we reviewed some of the most relevant smart contract fuzzing tools currently available. In addition, an empirical evaluation was conducted on their capability to correctly analyze interacting contracts and to handle large projects that require complex deployment steps. Our research points out that most of the existing tools fail in providing some important functionality, such as the ability to properly model interacting contracts and the ability to handle complex deployment steps. Therefore, even if these tools offer good performances and vulnerabilities detection rates, they are unlikely to be adopted in a real production scenario.

As future work, we plan to extend our analysis to other software sanitation techniques such as symbolic execution or static analysis.

REFERENCES

[1] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. 2020. Verification of smart contracts: a survey. *Pervasive and Mobile Computing*, 67, 101227.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 164–186.

[3] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: a scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*.

[4] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.

[5] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. 2023. A survey on smart contract vulnerabilities: data sources, detection and repair. *Information and Software Technology*, 107221.

[6] Confuzzius. https://github.com/christoftorres/ConFuzzius.

[7] Contractfuzzer. https://github.com/gongbell/ContractFuzzer.

[8] 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.

[9] Echidna. https://github.com/crytic/echidna.

[10] Ef/cf. https://github.com/uni-due-syssec/efcf-framework.

[11] Etheno. https://github.com/crytic/etheno.

[12] 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.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {Afl++}: combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[14] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 557–560.

[15] Dominik Harz and W Knottenbelt. 1809. Towards safer smart contracts: a survey of languages and verification methods (2018). *URL: http://arxiv. org/abs*.

[16] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 531–548.

[17] Ilf. https://github.com/eth-sri/ilf.

[18] Ityfuzz. https://github.com/fuzzland/ityfuzz.

[19] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 259–269.

[20] 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*, 254–269.

[21] 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.

[22] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. https://consensys.io/diligence/files/D1T2%20-%20Bernhard%20Mueller%20-%20Smashing%20Ethereum%20Smart%20Contracts%20for%20Fun%20and%20ACTUAL%20Profit.pdf.

[23] 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.

[24] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.

[25] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. 2023. Ef/cf: high performance smart contract fuzzing for exploit generation. *arXiv preprint arXiv:2304.06341*.

[26] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: attacks and protections. *IEEE Access*, 8, 24416–24427.

[27] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 322–333.

[28] Smartian. https://github.com/SoftSec-KAIST/Smartian.

[29] Xiangyan Tang, Ke Zhou, Jieren Cheng, Hui Li, and Yuming Yuan. 2021. The vulnerabilities in smart contracts: a survey. In *Advances in Artificial Intelligence and Security: 7th International Conference, ICAIS 2021, Dublin, Ireland, July 19-23, 2021, Proceedings, Part III 7*. Springer, 177–190.

[30] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54, 7, 1–38.

[31] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: a data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119.

[32] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*, 664–676.

[33] Understanding the dao attack. https://www.coindesk.com/learn/understanding-the-dao-attack/.

[34] Valentin Wüstholz and Maria Christakis. 2020. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1398–1409.