Towards an Ethereum Smart Contract Fuzz Testing Tool

Mariam Lahami[1][a], Moez Krichen[1,2][b], Mohamed Ali Mnassar[1], Racem Mrabet[1] and Mohamed Ben Rhouma[1]

[1]*ReDCAD Lab, National School of Engineers of Sfax, Sfax University, Sfax, Tunisia*

[2]*Faculty of CSIT, Al-Baha University, Al-Baha, Saudi Arabia*

Keywords: Fuzz Testing, Blockchain, Smart Contracts, Ethereum, Ganache, Brownie.

Abstract: The widespread and well-known blockchain platform that makes use of smart contracts is Ethereum. The key feature of these computer programs is that once deployed, they cannot be updated anymore. Therefore, it is highly necessary to efficiently test smart contracts before their deployment. This paper presents a Web-based testing tool called *LeoKai* that makes it easy to automatically generate test inputs and also unit test templates to detect bugs and vulnerabilities in Ethereum smart contracts. It helps developers to perform manual UI tests by invoking smart contracts deployed on the Ganache blockchain. Furthermore, it supports black-box fuzz testing and randomly generates test inputs. Blockchain developers may use the unit test template generator to generate unit tests. It also includes a code coverage module that highlights their efficiency by assessing function, branch, and line coverage. Finally, the prototype and its implementation details are illustrated.

1 INTRODUCTION

The last ten years have seen the emergence of blockchain technology, which has gained significant interest in various sectors (Krichen et al., 2022a), including supply chain management (Mars et al., 2021; Lahami and Chaabane, 2023), intelligent transportation (Jabbar et al., 2022), e-health (Fekih and Lahami, 2020; Abbas et al., 2021) and among others. In fact, blockchain is a distributed ledger initially proposed by Satochi Nakamoto (Nakamoto et al., 2008) as a linked chain of blocks in which transactions are securely stored. The essential features of this technology are decentralization, transparency, immutability, and security, which have boosted the interest in it. For example, the security is achieved by using cryptographic functions, while immutability is provided by sharing identical copies of the ledger among several peer-to-peer nodes.

The development of decentralized applications (dApps) has attracted a lot of attention in recent years, especially with the emergence of blockchain platforms like Ethereum[1] and Hyperledger Fabric[2]. Smart contracts (SC) are self-executing programs that are stored on the blockchain and are executed automatically when certain conditions are met without the need for third parties. They enable the implementation of business logic within the distributed ledger. Nevertheless, like any other computer program, they can contain bugs and vulnerabilities that might result in serious consequences, such as significant financial losses and data breaches. Consequently, checking smart contract correctness and improving trust in them by reducing the risk of losing money have to be taken into consideration.

As one of the most used methods to increase confidence in blockchain-oriented software and especially in smart contracts, fuzz testing is used in the literature (Akca et al., 2019; Jiang et al., 2018; Pan et al., 2021; Grieco et al., 2020; Liu et al., 2018). Fuzz testing is a black-box software testing technique that involves feeding a program with random, invalid, or unexpected inputs to detect bugs and vulnerabilities. The use of fuzz testing for smart contracts can greatly improve their security and reliability.

Several tools and test approaches have been introduced in the testing community to generate and execute effective test cases for smart contracts (Lahami et al., 2022; Maâlej and Lahami, 2023). In particular, fuzz testing has attracted a lot of research interest and

[a] https://orcid.org/0000-0002-8691-9848
[b] https://orcid.org/0000-0001-8873-9755
[1]https://ethereum.org/en/
[2]https://www.hyperledger.org/

Lahami, M., Krichen, M., Mnassar, M., Mrabet, R. and Ben Rhouma, M.
Towards an Ethereum Smart Contract Fuzz Testing Tool.
DOI: 10.5220/0012785400003753
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 19th International Conference on Software Technologies (ICSOFT 2024)*, pages 137-144
ISBN: 978-989-758-706-1; ISSN: 2184-2833
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

has been applied to smart contract testing. However, existing tools often require significant expertise and effort to use effectively and may not provide comprehensive testing coverage.

To address the need for reliable and effective smart contract testing, we present LeoKai, a web-based testing tool for Ethereum smart contracts. LeoKai aims to make it easy for developers to automatically generate test inputs and unit test templates, detect bugs and vulnerabilities, and perform Manual User Interface (UI) tests by invoking smart contracts deployed on the Ganache[3]blockchain. Additionally, LeoKai supports black-box fuzz testing by generating random test inputs and includes a code coverage module that assesses function, branch, and line coverage to highlight the efficiency of the testing process.

LeoKai is designed to be easy to use, even for developers with little experience in smart contract testing. It provides a user-friendly interface that allows developers to quickly generate and execute test cases, as well as analyze the results of their testing.

Overall, this paper presents LeoKai as a valuable tool for testing Ethereum smart contracts, providing developers with the means to identify and address potential bugs and vulnerabilities before deployment. By leveraging fuzz testing and unit testing templates, LeoKai can help improve the security and reliability of smart contracts and increase confidence in their functionality.

The remainder of this paper is organized as follows: Section 2 provides background materials for understanding blockchain features and introduces the fuzz testing technique. After that, we address related work in the context of black-box fuzzing in Section 3. Section 4 describes the architecture and design of LeoKai, including its components and how they interact to provide testing capabilities. Section 5 covers the implementation details and technologies used to develop LeoKai. Finally, we conclude in Section 6 and present potential areas of future research.

2 BACKGROUND MATERIALS

This section highlights the background materials that help to understand the rest of the paper: a brief discussion on topics related to Blockchain (BC), Smart Contracts (SCs), and fuzz testing concepts is presented.

2.1 Blockchain

A blockchain is a peer-to-peer system and a complex data structure that manages the flow of data without the need for a central authority. It was introduced by Nakamato et al. (Nakamoto et al., 2008) as the technology underlying Bitcoin. Blockchain is made up of a linked list of blocks. A chain is a linking mechanism that connects one block to another. This chaining is accomplished via cryptographic hashes. Each block carries a reference to the previous block's hash, resulting in a chronological and immutable chain of blocks. The block hash is obtained by executing a cryptographic hash function (e.g., SHA256, KECCAK256, etc.). Moreover, each block is composed of two parts: the header and the body. The header of a given block contains several fields, particularly a timestamp of when the block was produced and the hash of the previous block. In the body of the block, transaction details are stored, such as recipient, gas fees, asset, ownership, etc. Transactions are digitally signed instructions from accounts that update the Blockchain network's state. The most basic transaction is transferring crypto-currencies from one account to another.

It is worthy to note that we support the Ethereum blockchain in this work. The Ethereum network is the concept of blockchain's second evolution. It builds on the basic blockchain structure by adding numerous new programming languages. It has over 10,000 complete nodes and it is distributed internationally. Ethereum is largely utilized for the exchange of Ether and the creation of smart contracts.

2.2 Smart Contracts

Smart Contracts (SCs) are one of the most intriguing features developed by numerous platforms, including Ethereum and Hyperledger, with the goal of tying business logic code to transactions. A SC is defined as a piece of autonomous programming code that is deployed on the blockchain and executed when specific events occur.

Smart contracts are developed using various programming languages. In our work, we are using Solidity[4]: a Turing-complete language, which is quite close to JavaScript. It supports libraries, inheritance, and user-defined types. Solidity smart contracts are compiled to Ethereum Virtual Machine (EVM) bytecode using the Solidity compiler *solc*.

2.3 Potential Issues with Smart Contracts

Smart contrats may include several vulnerabilities and flaws (Krichen et al., 2022b; Praitheeshan et al.,

[3]https://trufflesuite.com/ganache/

[4]https://solidity.readthedocs.io/

2019). In the following, we introduce the most popular ones:

- Reentrancy: The DAO attack happened due to this vulnerability in smart contracts. A reentrancy occurs when a function invokes, through an external call, another suspicious contract. The latter then attempts to drain funds by calling the original function repeatedly.
- Unhandling Exception: There are numerous circumstances in which abnormal situations may occur, and this leads to trigger exceptions. However, these exceptions aren't usually handled in a consistent manner. This makes the contracts vulnerable to malicious attacks and leads to ether loss because Solidity programmers are not aware that these exceptions are not handled correctly and the transactions are reverted.
- Out-of-Gas Exception: Every transaction has a maximum amount of gas that can be spent, and thus it is considered as the amount of computation allowed, called the gas limit. The transaction will fail and an out of gas exception will be raised if the amount of gas used exceeds this limit. Also, this exception may occur when the function "send" to transfer ether to a contract is called.

2.4 Fuzz Testing

Fuzz testing, also known as fuzzing, is an established software testing technique that makes use of massive amounts of random data as input to find bugs, crashes, or security weaknesses in networks, operating systems, and traditional software (Tonella et al., 2014). Barton Miller of the University of Wisconsin invented this dynamic testing method in the late 1980s (Felderer et al., 2016). Since then, it has been recognized that fuzz testing is a useful and efficient method for identifying software vulnerabilities. While the initial approaches to fuzz testing relied only on randomly generated test data (i.e., random fuzzing), advances in model-based testing, symbolic computation, and dynamic test case generation led to improvements in fuzzing techniques like mutation-based fuzzing, generation-based fuzzing, or gray-box fuzzing. In the following, we introduce the most well-known fuzzing techniques in the literature (Felderer et al., 2016):

- **Random Fuzzing:** is the simplest fuzz testing technique. It consists in sending a stream of random input data, in a black-box scenario, to the program under test. This kind of fuzzing is very helpful for testing how a software responds to huge or erroneous input data. Thus, this technique is adopted in this work.
- **Mutation-Based Fuzzing:** This form of fuzzing requires the fuzzer to have some knowledge of the input format of the program being tested. Then, it makes use of existing data samples to build new versions (mutants), which it then uses for fuzzing.
- **Generation-Based Fuzzing:** A model (of the input data or the vulnerabilities) was utilized in this kind of fuzzing to generate test data from this model or specification. Compared to previous fuzzing techniques, it often provides an improved coverage of the program under test especially when the expected input format is complex.

3 RELATED WORK

In the last few years, several approaches and testing tools have been proposed in order to check the correctness of smart contracts. We study here the most related work to our proposal that makes use of the black-box fuzzing technique (Akca et al., 2019; Jiang et al., 2018; Grieco et al., 2020; Pan et al., 2021; Liu et al., 2018) and also unit testing (Motepalli et al., 2020; Medeiros et al., 2019).

3.1 Related Work on Fuzzing SCs

As already mentioned, Black-box fuzzing is a fundamental approach that produces random test data based on a distribution for various inputs. This method creates input for all kinds of programs by generating a bit stream representing the required data types.

Up to our best knowledge, ContractFuzzer is the first black-box fuzzer proposed in the literature for smart contract testing (Jiang et al., 2018). It takes as inputs the ABI and bytecode files generated by the Solidity compiler, with the aim of generating test inputs. Also, it defines test oracles for detecting seven security vulnerabilities in Ethereum smart contracts.

In the same direction, SolAnalyser (Akca et al., 2019) presents a vulnerability detection tool that follows three phases. First, this tool analyzes statically the source code of Solidity smart contracts in order to assess locations prone to vulnerabilities and then instrument it with assertions. After that, an *inputGenerator* module has been implemented to automatically generate inputs for all transactions and functions in the instrumented contract. Next, the presence of vulnerabilities is triggered when the property checks are violated during execution of smart contracts on the Ethereum Virtual Machine (EVM).

Similarly, authors in (Grieco et al., 2020) propose an open-source smart contract fuzzer automatically generating tests to detect assertion violations and some custom properties. The proposed library called Echidna creates test inputs depending on user-supplied predicates or test functions. The major issue with this tool is that it may require a lot of effort and great knowledge to write the predicates and test methods.

Furthermore, we discuss yet another interesting study (Pan et al., 2021) in which authors introduce a black-box fuzzer engine to generate inputs in order to detect reentrancy vulnerability. Called ReDefender, this tool would send transactions while gathering runtime data through fuzzing input. After that, ReDefender can identify reentrancy vulnerabilities and track the vulnerable functions by looking at the execution log. This tool demonstrates its ability to efficiently detect reentrancy bugs in real-world smart contracts.

Another interesting study was introduced in (Liu et al., 2018). The proposed tool called ReGuard exploits fuzz testing for smart contracts by iteratively producing various random and diverse transactions to detect reentrancy vulnerabilities in solidity-based smart contracts. ReGuard includes a fuzzing engine, which will generate iteratively random bytes using runtime coverage information as feedback.

3.2 Related Work on Unit Testing of SCs

Authors in (Medeiros et al., 2019) introduce an original approach for running smart contract unit tests that aims to speed up test execution. To do so, the proposal reuses the setup execution of each test as well as the deployment execution of the smart contract in each test to achieve this decrease. The developed framework in Java is called the SolUnit framework and is used to run Java tests for Ethereum Solidity smart contracts.

Following the same goal which the speed-up of test execution, FabricUnit, a novel framework for applying unit testing on Hyperledger Fabric, is proposed (Motepalli et al., 2020). First, this framework looks safe methods (For instance, Read operations are considered as safe methods) that not alter the state of the blockchain and then reuses the same test setup (@Before method which is in charge of cleaning up the environment and reinitializing it before each test execution) for all of them during the test execution phase. For unsafe methods, the test execution is started by running @Before methods. FabricUnit is implemented in Java and supports only Hyperledger Fabric applications with Go chaincode and Java client.

The work in (Olsthoorn et al., 2022) combines fuzzing and the generation of unit tests for solidity based-smart contracts. The proposed tool, called SynTest-Solidity, makes use of initial randomly generated test cases and genetic algorithms to efficiently and effectively test smart contracts. The produced test cases are written following the Mocha framework and executed by Truffle. The authors conducted an empirical study to evaluate the efficiency of their tool by validating twenty Solidity smart contracts.

Similar to this work, our approach includes a fuzzy engine to generate inputs for all functions in the smart contract under test. Its main advantage is that it combines manual UI testing, fuzzing, and unit test template generation. All these features help blockchain developers effectively and efficiently test smart contracts.

4 ARCHITECTURE AND DESIGN

Figure 1 shows the LeoKai architecture, which is divided into several modules with the aim of performing either manual UI testing of smart contract functions or fuzz testing while generating unit test templates to create an easy-to-use test tool for smart contract developers. Furthermore, we illustrate through a UML activity diagram the workflow that presents the adopted testing process (see Figure 2).

0. **AI-Auditor:** LeoKai incorporates an AI-based Solidity Smart Contract Auditor [5] to enhance the security analysis of the tested contracts. Leveraging machine learning techniques, the auditor employs a trained model to detect potential vulnerabilities and security risks in the contract code. By analyzing the contract's structure, control flow, and data flow, the AI-based auditor can identify common vulnerabilities, such as reentrancy, integer overflow/underflow, and unauthorized access issues. The auditor's ability to detect patterns and anomalies in the code significantly complements the fuzz testing process, providing an additional layer of security analysis and helping developers identify potential vulnerabilities that may not be captured through traditional testing approaches.
1. **Pre-Processing:** LeoKai checks whether the input file corresponds to a valid solidity file. Then, the compilation and deployment of the smart contract under test are performed. Note here that it is

[5]https://auditor.0x0.ai/

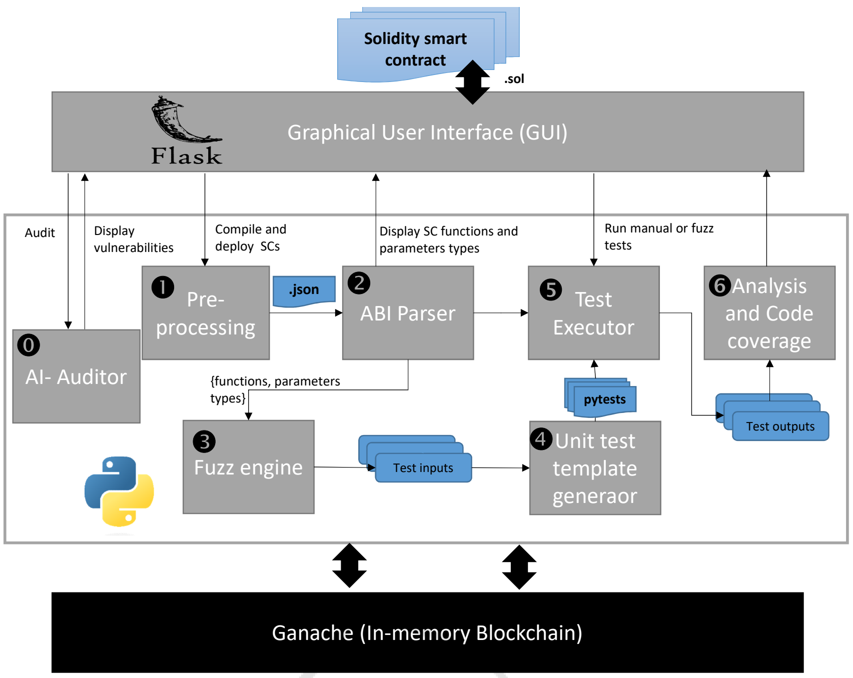

Figure 1: Overview of the LeoKai architecture.

required to make use of a development and testing framework for Ethereum smart contracts like Brownie or Truffle and also to launch an Ethereum blockchain simulator where the contracts can be deployed and tested, like Ganache.

2. **ABI Parser:** this module takes as input the Application Binary Interface (ABI) and bytecode files that are generated once the smart contract is compiled with the *solc* compiler. Indeed, the ABI file includes details about the functions in the contracts, such as their name, type (i.e., private, public, payable), argument types, etc. The bytecode file holds the predefined bytecode of the smart contract. These two files are required later to generate test inputs and execute them.

3. **Fuzz Engine:** LeoKai includes a black-box fuzz engine, which is responsible for automatically and efficiently generating test inputs. It takes the output of the previous step, including function names, arguments, argument types, etc. The proposed input generation algorithm may generate valid or invalid inputs for each function. We propose different strategies to generate inputs for fixed-size inputs and non-fixed-sized inputs. For example, we support fixed-size input types such as unsigned and signed integers with widths ranging from 8 to 256 bits (e.g., uint8, uint16). For the address type, we randomly generate one address from the ten addresses offered by the local blockchain platform. Algorithm 1 depicts our random fuzzing algorithm that takes as input a solidity-based smart contract and generates for each function a set of random inputs. The test input generation is based on the parameters of each function and their types (i.e., int, string, boolean, address or byte), as highlighted from Line 7 to Line 17. We used a map data structure to store the obtained inputs (see Line 18).

4. **Unit Test Template Generator:** this module focuses on generating unit test templates with the aim of making testing easier for developers. Unit tests are stored and can be updated and then executed.

5. **Test Executor:** We use the inputs created in step 3 to execute the contract, thus, each function within the contract is invoked at least once and generates a transaction. Moreover, it is possible to perform manual UI testing through the LeoKai GUI, and the developer may choose the function under test and the inputs used to invoke it.

6. **Analysis and Code Coverage:** The analysis and code coverage module is responsible for detecting the existence of any bug or vulnerability on the smart contract under test and also assessing the percentage of code coverage achieved during the test process. Indeed, LeoKai performs bug detection via transaction trace analysis. Concerning code coverage, it identifies functions and branches

Algorithm 1: Random Fuzzing Algorithm.

```
    Data: $smartContract.sol$
    Result: test inputs for each function in the
            smart contract are generated.
    /* Declare a map data structure in
       which the function is the key
       and its test inputs are the
       value                              */
1   $testInputsForFunctions \leftarrow null$;
2   $File\ artefactJson \leftarrow$
      $compile(smartContract)$;
3   $List\ lstFunction \leftarrow$
      $parseAbi(artefactJson.abi)$;
4   foreach $f \in lstFunction$ do
5       $testInputs.clear()$;
6       $params \leftarrow f.fetchParams()$;
7       foreach $p \in params$ do
8           if typeOf(p)isEqual(int) then
9               testInputs.add(randomInt(p.size()));
10          else if typeOf(p)isEqual(bool) then
11              testInputs.add(randomBool());
12          else if typeOf(p)isEqual(string) then
13              testInputs.add(randomString());
14          else if typeOf(p)isEqual(byte) then
15              testInputs.add(randomBool(p.size()));
16          else
17              testInputs.add(randomAdress());
18      $testInputsForFunctions[f] \leftarrow testinputs$;
```

that have been exercised during the test process. This information helps us evaluate the test's effectiveness and highlight potential gaps in coverage.

5 IMPLEMENTATION DETAILS

In this section, we present our Web-based test tool, *LeoKai*, which is essentially written in Python. As with any Web application, it is composed of two layers: the backend and the frontend. The backend layer forms the core of our test system and is responsible for executing the fuzz testing algorithm and interacting with the smart contracts. It utilizes the Brownie[6] framework, a Python-based development and testing framework for Ethereum smart contracts. By leveraging Brownie's functionalities, we can compile, deploy, and interact with smart contracts seamlessly. To compile the smart contracts, we utilize Brownie's built-in compiler, which supports multiple Solidity versions and automatically resolves dependencies. This ensures that our test system can handle various contract configurations and dependencies without manual intervention. The most relevant aspect offered by Brownie is that it allows blockchain developers to write and execute unit tests through the use of Pytest. Our tool extends this test framework by fuzzing capabilities and by generating unit test snippet codes.

Additionally, the backend layer interacts with the local blockchain Ganache, which provides a local test environment for deploying and executing the smart contracts. This allows us to perform testing activities without impacting the live Ethereum network or incurring real-world costs. It is connected to Ganache through *Web3.py* library. We utilize this library to create instances of the smart contracts, acting as proxies for function invocation and variable access. For fuzz testing, we systematically explore contract functions, generating random or semi-random inputs based on their signatures and argument types. These inputs are used to invoke functions through contract instances, covering various execution paths and corner cases. LeoKai captures executed transactions, input values, and changes in the contract's state for testing reports and analysis.

Note that the LeoKai backend was designed to work initially with Brownie and Ganache CLI, as they are popular among Smart Contract developers. Even so, it is possible to extend it to run with Truffle and Hardhat with some manual configurations.

Regarding the frontend layer, we utilize Flask[7], a lightweight Web framework, in order to provide a user-friendly interface for interacting with our application. This layer allows users to access and control features. Users can input the smart contract they want to test, configure the fuzzing parameters, and initiate the testing process. As highlighted in Figure 3, the initial screenshot of LeoKai shows some features offered by this tool.

Once the fuzz testing process is completed, comprehensive testing reports are generated to summarize the results and findings. These reports serve as valuable resources for developers, auditors, and stakeholders to assess the security and reliability of smart contracts. Moreover, a thorough test coverage analysis is conducted to provide insights into the areas of the contract that have been tested. It measures the percentage of code coverage achieved during the fuzz testing process, identifying functions and branches that have been exercised. This information helps us

[6]https://eth-brownie.readthedocs.io/en/stable/

[7]https://flask.palletsprojects.com/en/2.3.x/

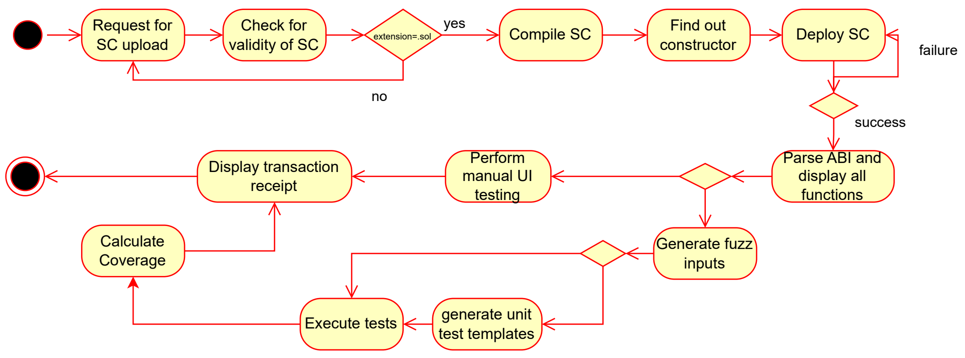

Figure 2: UML Activity diagram of LeoKai workflow.

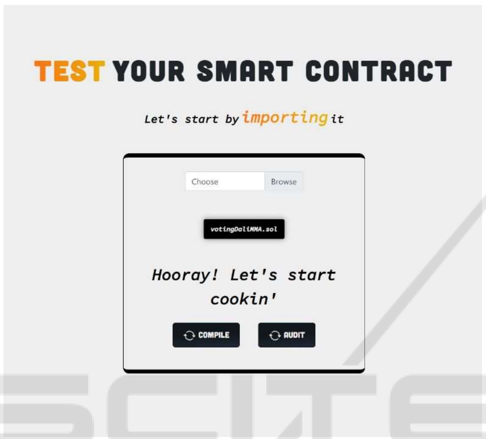

Figure 3: LeoKai initial screen.

evaluate the thoroughness of the testing and highlights potential gaps in coverage.

We have conducted a preliminary evaluation of LeoKai on several smart contracts from Etherscan and from the official documentation of Solidity. It shows that it is an effective tool for testing Ethereum smart contracts in terms of code coverage and bug detection.

6 CONCLUSION AND FUTURE WORK

In this work, we introduced LeoKai, a web-based testing tool that enables developers to quickly and effectively test Solidity smart contracts. By employing manual UI tests or fuzz tests and providing support for the generation of unit test templates written on PyUnit, LeoKai offers developers a comprehensive set of tools to identify and address potential bugs and vulnerabilities before deployment.

Our evaluation of LeoKai shows that it is an effective tool for testing Ethereum smart contracts in terms of code coverage and bug detection. However, there are several avenues for future work that we plan to pursue. Firstly, we aim to expand LeoKai's capabilities to support Vyper smart contracts and other popular contract development frameworks like Truffle, Embark, and Hardhat. Additionally, we envision generating unit test templates in Solidity to increase the flexibility of the tool. As a further step, we plan to create a plugin for the most popular code editors to make it even easier for developers to use LeoKai in their development workflow.

Furthermore, another potential future direction for enhancing the security of smart contracts is to incorporate artificial intelligence (AI) techniques into the testing process (Krichen, 2023). By leveraging the power of AI, it may be possible to significantly improve the efficiency and effectiveness of smart contract testing. AI can be used to automatically generate test cases, identify patterns in code that may indicate vulnerabilities, and even suggest fixes for detected issues. Integrating AI into smart contract testing could ultimately lead to more secure and reliable decentralized applications.

In conclusion, we believe that LeoKai is a valuable tool for testing Ethereum smart contracts, offering developers a comprehensive set of tools to identify and address potential bugs and vulnerabilities before deployment. We hope that our work will contribute to the broader effort to improve the security and reliability of decentralized applications and smart contracts on the Ethereum blockchain. As a future direction, we plan to explore the potential of AI in smart contract testing, which could significantly improve the efficiency and effectiveness of the testing process.

REFERENCES

Abbas, A., Alroobaea, R., Krichen, M., Rubaiee, S., Vimal, S., and Almansour, F. M. (2021). Blockchain-assisted secured data management framework for health information analysis based on internet of medical things. *Personal and Ubiquitous Computing*, pages 1–14.

Akca, S., Rajan, A., and Peng, C. (2019). Solanalyser: A framework for analysing and testing smart contracts. In *Proceeding of the 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489.

Fekih, R. B. and Lahami, M. (2020). Application of blockchain technology in healthcare: A comprehensive study. In *Proceeding of 18th International Conference of The Impact of Digital Technologies on Public Health in Developed and Developing Countries, ICOST 2020, Hammamet, Tunisia, June 24-26*, pages 268–276.

Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., and Pretschner, A. (2016). Chapter one - security testing: A survey. volume 101 of *Advances in Computers*, pages 1–51. Elsevier.

Grieco, G., Song, W., Cygan, A., Feist, J., and Groce, A. (2020). Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 557–560, New York, NY, USA. Association for Computing Machinery.

Jabbar, R., Dhib, E., Said, A. B., Krichen, M., Fetais, N., Zaidan, E., and Barkaoui, K. (2022). Blockchain technology for intelligent transportation systems: A systematic literature review. *IEEE Access*, 10:20995–21031.

Jiang, B., Liu, Y., and Chan, W. K. (2018). Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, page 259–269.

Krichen, M. (2023). Strengthening the security of smart contracts through the power of artificial intelligence. *Computers*, 12(5).

Krichen, M., Ammi, M., Mihoub, A., and Almutiq, M. (2022a). Blockchain for modern applications: A survey. *Sensors*, 22(14).

Krichen, M., Lahami, M., and Al-Haija, Q. A. (2022b). Formal methods for the verification of smart contracts: A review. In *Proceedings of the 15th International Conference on Security of Information and Networks, SIN 2022, Sousse, Tunisia, November 11-13, 2022*, pages 1–8. IEEE.

Lahami, M. and Chaabane, F. (2023). Improving the supply chain management via blockchain: an olive oil case study. In Kallel, S., Benzadri, Z., and Kacem, A. H., editors, *Proceedings of the Tunisian-Algerian Joint Conference on Applied Computing (TACC 2023), Sousse, Tunisia, November 8-10, 2023*, volume 3642 of *CEUR Workshop Proceedings*, pages 182–193. CEUR-WS.org.

Lahami, M., Maâlej, A. J., Krichen, M., and Hammami, M. A. (2022). A comprehensive review of testing blockchain oriented software. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2022, Online Streaming, April 25-26, 2022*, pages 355–362. SCITEPRESS.

Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., and Roscoe, B. (2018). Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68.

Maâlej, A. J. and Lahami, M. (2023). White-box mutation testing of smart contracts: A quick review. In Hedia, B. B., Maleh, Y., and Krichen, M., editors, *Proceedings of 16th International Conference of Verification and Evaluation of Computer and Communication Systems, VECoS 2023, Marrakech, Morocco, October 18-20, 2023*, volume 14368 of *Lecture Notes in Computer Science*, pages 135–148. Springer.

Mars, R., Youssouf, J., Cheikhrouhou, S., and Turki, M. (2021). Towards a blockchain-based approach to fight drugs counterfeit. In *Proceedings of the Tunisian-Algerian Joint Conference on Applied Computing (TACC 2021), Tabarka, Tunisia*, pages 197–208.

Medeiros, H., Vilain, P., Mylopoulos, J., and Jacobsen, H.-A. (2019). Solunit: A framework for reducing execution time of smart contract unit tests. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 264–273, USA. IBM Corp.

Motepalli, S., Vilain, P., and Jacobsen, H.-A. (2020). Fabricunit: A framework for faster execution of unit tests on hyperledger fabric. In *Proceeding of the IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3.

Nakamoto, S. et al. (2008). Bitcoin: A peer-to-peer electronic cash system.

Olsthoorn, M., Stallenberg, D., Van Deursen, A., and Panichella, A. (2022). Syntest-solidity: Automated test case generation and fuzzing for smart contracts. In *Proccedings of the IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 202–206.

Pan, Z., Hu, T., Qian, C., and Li, B. (2021). Redefender: A tool for detecting reentrancy vulnerabilities in smart contracts effectively. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 915–925.

Praitheeshan, P., Pan, L., Yu, J., Liu, J. K., and Doss, R. (2019). Security analysis methods on ethereum smart contract vulnerabilities: A survey. *CoRR*, abs/1908.08605.

Tonella, P., Ricca, F., and Marchetto, A. (2014). Chapter 1 - recent advances in web testing. In Memon, A., editor, *Advances in Computers*, volume 93 of *Advances in Computers*, pages 1–51. Elsevier.