SolSEE: A Source-Level Symbolic Execution Engine for Solidity

Shang-Wei Lin
shang-wei.lin@ntu.edu.sg
Nanyang Technological University, Singapore

Palina Tolmach
palina001@ntu.edu.sg
Institute of High Performance Computing, A*STAR
Nanyang Technological University, Singapore

Ye Liu
li0003ye@ntu.edu.sg
Nanyang Technological University, Singapore

Yi Li
yi_li@ntu.edu.sg
Nanyang Technological University, Singapore

ABSTRACT

Most of the existing smart contract symbolic execution tools perform analysis on bytecode, which loses high-level semantic information presented in source code. This makes interactive analysis tasks—such as visualization and debugging—extremely challenging, and significantly limits the tool usability. In this paper, we present SolSEE, a source-level symbolic execution engine for Solidity smart contracts. We describe the design of SolSEE, highlight its key features, and demonstrate its usages through a Web-based user interface. SolSEE demonstrates advantages over other existing source-level analysis tools in the advanced Solidity language features it supports and analysis flexibility. A demonstration video is available at: https://sites.google.com/view/solsee/.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; **Software verification and validation**.

KEYWORDS

Smart contract, symbolic execution

ACM Reference Format:
Shang-Wei Lin, Palina Tolmach, Ye Liu, and Yi Li. 2022. SolSEE: A Source-Level Symbolic Execution Engine for Solidity. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558923

1 INTRODUCTION

Symbolic execution is a program analysis technique which explores multiple execution paths of a program by assigning symbolic—instead of concrete—values to variables. For each analyzed execution path, a *symbolic execution engine* maintains (1) a *path condition*—a first-order Boolean formula that describes the conditions satisfied by the branches taken along that path, and (2) a *symbolic memory store* that maps variables to symbolic expressions or values. An off-the-shelf constraint solver is typically used to determine the feasibility of each explored path based on the generated path condition formula [3].

Symbolic execution is commonly used to systematically explore the program space and detect property violations as well as security vulnerabilities. In recent years, symbolic execution has been extensively applied on smart contracts—computer programs that run on blockchain and govern billions of dollars [19], which brings paramount importance to the security and correctness of the contract code [27]. Most smart contracts are written in Solidity [1]—a high-level programming language of the Ethereum blockchain platform, which compiles into EVM bytecode for deployment and execution on the blockchain. Listing 1 shows a sample Solidity implementation of a token smart contract. In Ethereum, users interact with a smart contract by initiating transactions that invoke its functions with `public` or `external` visibility. `Token`'s `deposit()` function (lines 18–20) allows the contract to accept ETH, a native cryptocurrency of Ethereum, and records the deposited amount (`msg.value`) in a `balances` mapping entry associated with a transaction sender (`msg.sender`). The `recover()` function (lines 21–23) sends all ETH balance of `Token` to its `owner`, the address that performed contract deployment (i.e., creation) which involved the execution of a `constructor` function (lines 14–17). Access control on `recover()` is implemented using the functionality of the `Ownable` smart contract (lines 1–10) that `Token` inherits from (line 11). Thus, the signature of `recover()` (line 21) contains an invocation of the `onlyOwner` modifier (lines 7–8). The modifier adds additional behavior, such as a prerequisite (line 7), to a function body which replaces "`_;`" in a modifier code (line 8). The transfer of ETH (line 22) is performed via Ethereum's built-in `.call.value()()` function call.

Given a smart contract written in Solidity, SolSEE symbolically represents the (storage/memory) configuration of the smart contract and executes each statement based on the operational semantics of Solidity [1, 8]. Our developed operational semantics for Solidity supports many important features including inheritance, modifiers, ETH transfer, and others. During symbolic execution, this representation is directly used to determine satisfiability of the generated path constraints using Z3 SMT-solver [4]. To facilitate efficient exploration of interesting smart contract behaviors in a realistic setting, SolSEE supports user-defined *harness* function that specifies the sequence of function calls to be analyzed symbolically. The harness definition follows exactly the syntax and semantics of Solidity, which is intuitive and easy to use for developers.

Listing 2 illustrates how to define a harness in SolSEE. The `_MAIN_` contract serves as the entry point (similar to the `main()`

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558923

```
1  contract Ownable {
2     address internal owner;
3     constructor(address _owner) public {
4        owner = _owner;
5     }
6     modifier onlyOwner() {
7        require(owner == msg.sender);
8        _;
9     }
10 }
11 contract Token is Ownable {
12    mapping (address => uint256) public balances;
13    string public name;
14    constructor(string memory _tokenName) Ownable(msg.
         sender) public payable {
15       name = _tokenName;
16       deposit();
17    }
18    function deposit() public payable {
19       balances[msg.sender] += msg.value;
20    }
21    function recover() public onlyOwner {
22       owner.call.value(address(this).balance)("");
23    }
24 }
```

Listing 1: Source code of the Token smart contract

```
1  contract _MAIN_ {
2     Token token; bool fallbackExecuted;
3     constructor () public {
4        uint $amount;
5        __assume__(address(this).balance >= $amount);
6        token = (new Token).value($amount)("OT");
7        token.recover();
8        assert(fallbackExecuted);
9     }
10    function() external payable {
11       fallbackExecuted = true;
12    }
13 }
```

Listing 2: Solidity source code of the harness contract

function in C), and the harness is defined as a constructor of it. The harness contains the declaration of a symbolic variable which should come with the prefix "$" (line 4) and the creation of the `Token` smart contract which involves transferring a symbolic amount of ETH to this contract (line 6). The harness also includes a call to the `Token`'s `recover()` function, which is executed successfully, because the call is invoked by the `_MAIN_` contract, satisfying the `onlyOwner` modifier. Lines 10–12 show the definition of a *fallback* function—another Solidity-specific feature supported by SolSEE. Here, the fallback function is invoked when ETH is transferred to a smart contract, i.e., upon the execution of the "`.call.value()()`" in Listing 1, line 22. Since the execution of `recover()` succeeds, the assertion in Listing 2, line 8, should always be satisfied. The assertion in a harness function can also be used to declare a property, which can be constructed using Solidity operators and variables available in the `_MAIN_` contract. In addition, SolSEE allows assumptions to be specified in terms of smart contract's variables using the `__assume__()` statement. For example, line 5 in Listing 2 indicates that the analysis will only be concerned with the paths where the ETH balance of the `_MAIN_` contract is not less than the amount of ETH being sent to `Token` upon its creation.

Related Work. Most of the existing symbolic execution tools for smart contracts operate on bytecode (rather than source-code) level, which retains limited semantic information about the smart contract and, hence, complicates reasoning about high-level properties of a smart contract. Most of these tools focus on detecting well-known vulnerabilities based on a certain pattern appearing in smart contract bytecode, e.g., Oyente [14], Mythril [17], Maian [18], etc. Manticore [16] is a bytecode-level symbolic execution engine which supports property-based symbolic execution and provides users with some control over the state exploration process.

While Manticore also offers a GUI plugin, it visualizes low-level bytecode instructions, which are difficult, for a developer, to match with their original Solidity source code statements. Meanwhile, existing source-level tools for Solidity smart contracts offer limited support for Solidity features and/or do not allow customization of the function call sequence and the environment to be analyzed. For example, VeriSmart [23] is a smart contract verifier that is also used in SmarTest [22] to perform symbolic execution of smart contracts. These tools do not precisely handle the execution of fallback functions and inter-contract function calls, which constitute essential functionality of smart contracts. Inter-contract function calls are also not analyzed precisely by SMTChecker [2, 5]—a built-in verifier within the Solidity compiler. Two other source-level tools, solc-verify [7] and VeriSol [30], translate Solidity code into Boogie intermediate language, which can introduce discrepancy between the analyzed code translation and original Solidity semantics. In addition, they also lack support of certain Solidity functionality and do not allow customization of the harness function and analyzed environment, which leads to multiple false positives reported by these tools, as shown in Sect. 3. ESBMC-Solidity [24] translates Solidity into an intermediate representation of ESBMC, which may introduce semantics discrepancy, and it does not support certain Solidity features such as polymorphism and inheritance.

Contribution. In this demonstration paper, we present SolSEE, a user-friendly symbolic execution engine for analyzing source code of one or several interacting smart contracts written in Solidity. The key features of SolSEE can be summarized as follows:

- **Precise operational semantics.** SolSEE symbolically represents the configuration of smart contracts and executes each program statement based on the exact operational semantics for Solidity version 0.5.
- **User-defined harness function.** SolSEE facilitates analysis and debugging of smart contracts by allowing users to define the harness function to control the function call sequence for verification. SolSEE detects and reports unsigned integer under- and overflow and checks the validity of assertions, which can be used to specify custom high-level properties of the analyzed smart contracts.
- **Smart contract debugging.** With the symbolic paths generated by SolSEE, users can debug smart contracts. Users are able to visualize the execution details corresponding to the symbolic paths step by step in a Web user interface.

2 METHODOLOGY

In this section, we introduce the design and usage of SolSEE.

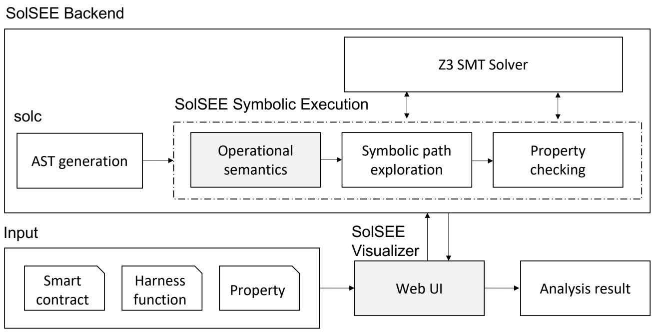

Figure 1: Tool architecture.

2.1 Design

SolSEE was implemented with 7,521 lines of C++ code and currently supports smart contracts written in Solidity v0.5. Figure 1 demonstrates an overview of the tool architecture. SolSEE takes one or more smart contracts and a harness function as input, which should be provided using a Web UI. In its backend, SolSEE uses a Solidity compiler solc to generate the AST for the given contracts. Then, SolSEE symbolically executes each statement by traversing the AST based on the Solidity operational semantics. The (storage/memory) configuration of the smart contract is encoded symbolically as Z3 types [4]. Also, the SMT solver is used to discharge the feasibility queries of symbolic paths and validity queries of assertions. The latter helps prove or disprove the user-defined properties encoded as assertions in the harness function. SolSEE does not bound the loop iterations and may unroll the loops infinitely. This can be addressed by requesting the user to provide a loop invariant or generating one automatically [11, 12], which is left for future work.

The frontend of SolSEE is based on the Remix IDE [21], which is implemented as a ReactJs [20] application. The frontend of SolSEE communicates with its backend via restful APIs. SolSEE also provides a debugging capability and helps developers examine the operation of a smart contract by visualizing its (symbolic) execution in detail. Its usage is presented in Sect. 2.2.

The symbolic execution module of SolSEE supports a majority of the Solidity language features, including the intra- and inter-contract function calls, multiple inheritance, library support via the "`using ... for`" construct, low-level function calls such as "`.call.value()()`" with the associated fallback mechanism, modifiers, and many others. Similarly to the Solidity compiler, SolSEE automatically generates getter functions for public smart contract variables of elementary types. Similar to other source-level analyzers for smart contracts [7, 23, 30], SolSEE does not support inline assembly. SolSEE also introduces a supplementary `__assume__()` statement that can be used to specify assumption conditions when verification is performed, as shown in Listing 2, line 5.

In SolSEE, `require()` and `assert()` have different semantics, although in Solidity, both functions could lead to a transaction with all its effects on the state being reverted, if the required/asserted condition is not satisfied. In Solidity, `require()` is used to check a condition that is expected to fail occasionally, e.g., a guard condition on function input arguments. Thus, should the expression enclosed in `require()` evaluate to *false*, SolSEE rollbacks all effects of the transaction on the smart contract state. We consider each statement in the harness function as one transaction. Semantics of `assert()` correspond to its purpose in Solidity: it is used to check conditions that should never evaluate to *false*. SolSEE stops execution and reports a violation if the asserted condition is violated.

In addition, SolSEE also reports possible integer under- and overflows—a common issue in Solidity smart contracts, which heavily utilize unsigned integers to store important information such as token balances [28]. SolSEE takes a modular arithmetic approach to handling unsigned integers of various sizes (from `uint8` to `uint256`): it models them using Z3 integers constrained by range assertions to follow the semantics of unsigned integer arithmetic operations in Solidity. Although using bitvectors is another popular approach to model unsigned integers, it has been shown to have scalability issues. To model one- and multi-dimensional arrays and mappings, SolSEE relies on the array theory.

The symbolic execution process of a smart contract is guided by the harness function provided by the user to orchestrate the interactions with the analyzed smart contract(s). During path exploration and assertion/property checking, SolSEE relies on Z3 [4], an SMT-solver, to resolve constraints. Using a harness function makes symbolic analysis performed by SolSEE highly configurable, which is necessary to effectively and efficiently analyze complex smart contract code in a realistic setting, which is demonstrated by our evaluation shown in Sect. 3. Additionally, a harness can also be used to encode properties about the execution trace or smart contract invariants in a form of assertions. To optimize tool performance, smart contract variables in a harness or analyzed smart contracts are assumed to have concrete (default) values unless they are declared as symbolic. Ethereum balance of a harness (`_MAIN_`) smart contract is assumed to be symbolic too.

2.2 Usage

SolSEE has both a command-line interface and a GUI. Given a file that contains Solidity source code of all smart contracts to be analyzed, e.g., `Token.sol`, symbolic execution via SolSEE can be invoked using the following command:

```
./SolSEE -symexe-main ./Token.sol
```

Figure 2 shows the Web GUI of SolSEE. The UI is built on the Remix IDE framework and allows users to do the following:

(1) Develop smart contracts in the "Smart Contract Panel";
(2) Customize the `_MAIN_` contract serving as the harness for analysis and verification in the "Harness Contract Panel";
(3) Click on the "Symbolic Execution Button" to trigger the symbolic execution of SolSEE to obtain a set of symbolic paths;
(4) Visualize the detail of each symbolic path in the "Result Panel";
(5) Click on the "Debugging Button" for further debugging.

The detailed description of each step can be found in the Appendix.

In SolSEE, we consider a small-step operational semantics. Thus, if a statement includes several function calls, users need to separate function calls into different statements. For example, the following statement with two function calls: `a = f() + g();` can be rewritten into three statements: `t1 = f();   t2 = g();   a = t1 + t2;`. While it is not convenient in syntax, it forces the developer to explicitly specify the order of function evaluation, which is not specified in the official Solidity document [6]. This enforcement eliminates ambiguity for analysis and verification, especially when

Table 1: Evaluation results.

Smart contract	Feature	SolSEE	VeriSol	VeriSol-SP	solc-verify	VeriSmart	SMTChecker
Token	Token implementation	✓	✗	✗	✗	✗	✗
MultipleModifiers	Two modifiers applied to one function	✓	✗	✗	✗	✓*	✓*
FallbackFunction	Fallback function execution	✓	✗	✗	✗	✗	✗
GetterFunction	Auto-generated getter functions	✓	✓	✓	✗	✗	✗
SafeMathLibrary	Using library functions on `uint`	✓	✗	✓	✗	✓*	✓*
MultipleInheritance	Multiple inheritance via C3 linearization	✓	✗	✓	✗	✗	✗
Structs	Arithmetic operations on `struct`	✓	✓	✓	✓	✓	✗
NewBytesArray	New dynamic memory array of `bytes32`	✓	✗	✗	✗	✗	✗
UintOverflow	`uint8` overflow detection	✓	✓†	✓†	✓†	✗	✓
Revert	Proper handling of `revert()` in a function	✓	✗	✗	✗	✓*	✗

* The code is analyzed correctly only in the absence of external function calls; † Non-default modular arithmetic mode must be used

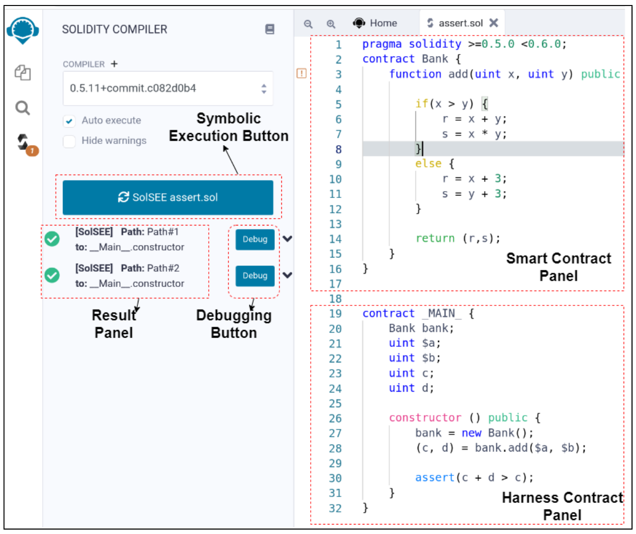

Figure 2: The Web user interface of SolSEE.

both `f()` and `g()` have side effects on state variables, and different execution orders may lead to different results.

3 EVALUATION

In this section, we demonstrate the capabilities of SolSEE and compare it with other source-level tools for Solidity smart contracts. The tools we compare with include solc-verify [7, 25], VeriSol v0.1.5 [15, 30] and its modified version used in SmartPulse [26, 29] (denoted as VeriSol-SP in Table 1), VeriSmart [10, 23], and SMTChecker [5] included in solc v.0.5.11. All these tools claim that they can identify assertion failures, which is the capability we perform the evaluation on. We could not run ExGen [9] due to compilation issues. ESBMC-Solidity [24] cannot process smart contracts even from its own documentation, so we do not compare with it as well. We evaluated SolSEE and these tools on our running example (Listing 1) and a dataset of nine different features present in Solidity. Table 1 summarizes how these features are supported by different tools. Each of the nine features has one corresponding smart contract, as shown in the first two columns of Table 1.

Our results demonstrate that all tools used for comparison lack support of certain Solidity features, except SolSEE. For example, VeriSol does not support modifiers with multiple "_;" statements (witness: `MultipleModfiers`). VeriSmart cannot handle fallback functions (witness: `FallbackFunction`) or correctly analyze arithmetic operations on variables of unsigned integer (`uint`) types (witness: `UintOverflow`). solc-verify and VeriSol follow the semantics of arithmetic operations for `uint` in Solidity, only if a non-default modular arithmetic mode is enabled. solc-verify fails to process a `bytes32` array in `NewBytesArray`, while variables of type `struct` are not supported by SMTChecker (witness: `Structs`). In addition, our results show that the analyzed tools report potential violation of numerous assertions, which are, in fact, false positives (e.g., solc-verify supports most Solidity features used in experimental smart contracts, but in many of them it reports all assertions as potentially failing). We attribute this fact to the lack of harness function support and missing or incorrect handling of Solidity language features and their semantics. For example, none of these tools, except SolSEE and SmartPulse , correctly implement C3 Linearization that Solidity uses to decide the order in which methods are inherited in the presence of multiple inheritance (witness: `MultipleInheritance`). Besides, while VeriSmart and SMTChecker process some of our smart contracts correctly, they can only do so if all the functionality is stored in a single contract, i.e., no external function calls allowed. In terms of the speed of analysis, our tool is as efficient as other tools used for comparison.

In closing, SolSEE has a unique source-level GUI that visualizes the symbolic execution process, which facilitates debugging.

4 CONCLUSION

This paper presents SolSEE, a user-friendly symbolic execution engine for smart contracts written in Solidity. SolSEE offers a large degree of customization which enables highly effective symbolic analysis of real-world smart contracts under realistic settings. Our evaluation shows that SolSEE is useful in analyzing interacting smart contracts in an efficient manner based on the user-defined harness function and assertions. SolSEE also facilitates analysis and debugging of Solidity smart contracts through a source-level visualization of symbolic execution in a GUI.

5 DATA AVAILABILITY

The binary of SolSEE and experimental smart contract dataset used for the evaluation are available on our website [13]. The release of SolSEE source code is pending approval from the funding agency.

REFERENCES

[1] 2020. Solidity — Solidity 0.5.11 documentation. https://solidity.readthedocs.io/en/v0.5.11/. Accessed: June 19, 2022.

[2] Leonardo Alt and Christian Reitwiessner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 376–388.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[5] Ethereum. 2022. SMTChecker and Formal Verification. https://docs.soliditylang.org/en/latest/smtchecker.html. Accessed: June 19, 2022.

[6] Ethereum. 2022. Solidity in Depth — Solidity 0.5.11 Documentation. https://docs.soliditylang.org/en/v0.5.11/solidity-in-depth.html. Accessed: June 19, 2022.

[7] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. *CoRR* abs/1907.04262 (2019). arXiv:1907.04262 http://arxiv.org/abs/1907.04262

[8] J. Jiao, S. Kan, S. Lin, D. Sanan, Y. Liu, and J. Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1265–1282. https://doi.org/10.1109/SP40000.2020.00066

[9] Ling Jin, Yinzhi Cao, Yan Chen, Di Zhang, and Simone Campanoni. 2022. EXGEN: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–1. https://doi.org/10.1109/TDSC.2022.3141396

[10] kupl. 2022. Source code of VeriSmart. https://github.com/kupl/VeriSmart-public/. Accessed: June 19, 2022.

[11] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic Loop-invariant Generation and Refinement through Selective Sampling. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 782–792.

[12] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, and Henri Hansen. 2017. FiB: Squeezing Loop Invariants by Interpolation between Forward/Backward Predicate Transformers. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 793–803.

[13] Shang-Wei Lin, Palina Tolmach, Ye Liu, and Yi Li. 2022. SolSEE: Online Supplementary Material. https://sites.google.com/view/solsee/. Accessed: May 26, 2022.

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

[15] Microsoft. 2022. Source code of VeriSol — A formal verifier and analysis tool for Solidity Smart Contracts. https://github.com/microsoft/verisol. Accessed: June 19, 2022.

[16] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. 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)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[17] Bernhard Mueller. 2018. *Smashing Ethereum Smart Contracts for Fun and Real Profit*. Technical Report.

[18] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), 653–663. https://doi.org/10.1145/3274694.3274743

[19] DeFi Pulse. 2022. DeFi - The Decentralized Finance Leaderboard at DeFi Pulse. https://defipulse.com/. Accessed: May 26, 2022.

[20] ReactJs. 2022. React - A JavaScript library for building user interfaces. https://reactjs.org/. Accessed: June 19, 2022.

[21] Remix. 2022. Remix IDE. https://github.com/ethereum/remix-project. Accessed: June 19, 2022.

[22] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1361–1378. https://www.usenix.org/conference/usenixsecurity21/presentation/so

[23] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2019. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. https://doi.org/10.48550/ARXIV.1908.11227

[24] Kunjian Song, Nedas Matulevicius, Eddie B. de Lima Filho, and Lucas C. Cordeiro. 2022. ESBMC-Solidity: An SMT-Based model checker for Solidity smart contracts. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 65–69. https://doi.org/10.1109/ICSE-Companion55297.2022.9793786

[25] SRI-CSL. 2022. Source Code of solc-verify—a modular verifier for Solidity. https://github.com/SRI-CSL/solidity/tree/boogie/. Accessed: June 19, 2022.

[26] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/sp40001.2021.00085

[27] Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. 2021. Formal Analysis of Composable DeFi Protocols. In *Financial Cryptography and Data Security. FC 2021 International Workshops*. Springer Berlin Heidelberg, Berlin, Heidelberg, 149–161.

[28] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (jul 2021), 38 pages. https://doi.org/10.1145/3464421

[29] utopia group. 2022. Source code of VeriSol used in SmartPulse. https://github.com/utopia-group/verisol. Accessed: June 19, 2022.

[30] Yuepeng Wang, Shuvendu Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2019. Formal Specification and Verification of Smart Contracts for Azure Blockchain. (April 2019). https://www.microsoft.com/en-us/research/publication/formal-specification-and-verification-of-smart-contracts-for-azure-blockchain/