ESBMC-Solidity: An SMT-Based Model Checker for Solidity Smart Contracts

Kunjian Song
Nedas Matulevicius
University of Manchester
Manchester, UK
kunjian.song@postgrad.manchester.ac.uk
nedas.matulevicius@postgrad.manchester.ac.uk

Eddie B. de Lima Filho
TPV Vision Innovator
Manaus, Brazil
eddie_batista@yahoo.com.br

Lucas C. Cordeiro
University of Manchester
Manchester, UK
lucas.cordeiro@manchester.ac.uk

ABSTRACT

Smart contracts written in Solidity are programs used in blockchain networks, such as Etherium, for performing transactions. However, as with any piece of software, they are prone to errors and may present vulnerabilities, which malicious attackers could then use. This paper proposes a solidity frontend for the efficient SMT-based context-bounded model checker (ESBMC), named ESBMC-Solidity, which provides a way of verifying such contracts with its framework. A benchmark suite with vulnerable smart contracts was also developed for evaluation and comparison with other verification tools. The experiments performed here showed that ESBMC-Solidity detected all vulnerabilities, was the fastest tool and provided a counterexample for each benchmark. A demonstration is available at https://youtu.be/3UH8_1QAVN0.

CCS CONCEPTS

• **Software and its engineering** → **Model checking**; **Software verification**.

KEYWORDS

Formal Verification, Solidity

ACM Reference Format:
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 *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516855

1 INTRODUCTION

The blockchain system is a distributed ledger technology that forms the primary mechanism behind Bitcoin, Ethereum, and alternative cryptocurrencies [1]. It can be considered as a singly linked list of blocks [2], where each of them contains a set of unmodifiable transactions. This way, such a technology serves as a distributed tamper-resistant record of such transactions [3]. Ethereum, for instance, can be regarded as a state machine whose global state is updated by those transactions, which indeed constitute state transitions.

The transactions are performed by smart contracts, which are programs automatically executed on blockchain networks when specific conditions are met [1], which encode business logic. For instance, such conditions can be an exchange of cryptocurrency or even a process of content unlocking if a digital rights management system is involved. Indeed, transactions act as stimuli to smart contracts. Nevertheless, such contracts must first be written in a given language. In the case of Etherium, smart contracts are written in Solidity, which is an object-oriented language for programs to be run on the Ethereum virtual machine (EVM) [2].

Once deployed, there is no way to update a smart contract except for deleting it entirely and re-deploying a new one. Even a smart contract's author cannot modify the corresponding source code or fix bugs after that [4]. Due to such immutability, it is critical to ensure that a smart contract is secure before its deployment on a blockchain network, such as Ethereum. However, as usually happens to software, smart contracts suffer from vulnerabilities, which represent a risk as malicious attackers often exploit them. As an example, the DAO attack, in 2016, resulted in a monetary loss of more than $50 million dollars, which forced Ethereum to be hard forked and then rolled back to a previous state [5].

If there can be vulnerabilities, software testing becomes paramount, and the community has already begun to tackle the related problems [6, 7]. However, most approaches target only a limited number of errors, which worsens as new applications appear and the need for specific aspects arises. Consequently, it is essential to employ mature and flexible verifiers, e.g., based on model checking and satisfiability modulo theories (SMT), to check smart contracts. This way, a myriad of problems is already handled, and consequent methodology advancements can be devised faster, including behavior models and specific properties [8].

In that sense, the efficient SMT-based context-bounded model checker (ESBMC) is a good candidate [9, 10]. It is a state-of-the-art (SOTA) checker, which can be extended to support different programming languages and target systems, such as digital filters and controllers, even incorporating behavior models and companion tools [8, 11, 12]. In addition, it was initially devised as a C-language model checker and has been evaluated using standard benchmarks and embedded applications in the telecommunication industry [13–15]. Recent efforts include the development of a new frontend to verify the most recent C++ standard [16].

This paper tackles the problem raised here, i.e., smart-contract verification, and proposes a frontend for ESBMC based on the newly

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516855

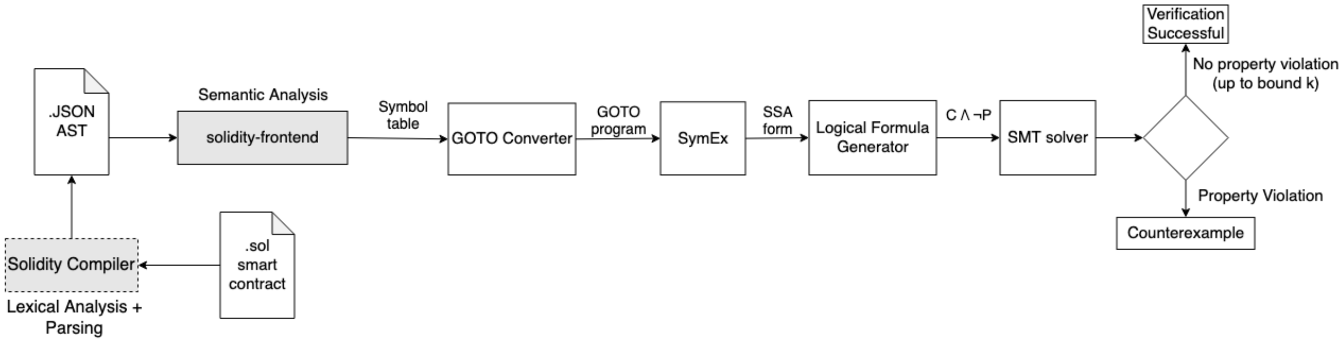

Figure 1: Architectural overview of ESBMC with its extension for verifying Solidity smart contracts. The new frontend takes Solidity JSON AST as input and converts the AST into ESBMC's IR, which is used to generate the symbol table. It then generates the GOTO program, which is symbolically executed to generate the program SSA form. Finally, the logical equations to represent the constraints (C) and properties (P) are created from the SSA form. The SMT solver finds the satisfiability of the equations only when there is a violation of the safety property; otherwise, the smart contract is considered safe up to the bound k.

developed Grammar-based Hybrid Conversion methodology explicitly developed for this work. It enables ESBMC to verify Solidity contracts, written in the Solidity language, using its available tools and techniques via two steps. First, we convert Solidity JavaScript object notation (JSON) abstract syntax trees (AST) into the ESBMC's intermediate representation (IR). Second, we integrate with ESBMC's infrastructure middleware and backend to reuse its existing SMT-based verification strategies (incremental and k-induction [17]).

In order to evaluate the proposed framework, named ESBMC-Solidity, a benchmark suite with vulnerable smart contracts was created and used as input for it and other SOTA Solidity verification tools: Smartcheck [18], Slither [19], Oyente [20], and Mythril [21]. ESBMC-Solidity outperformed the mentioned verification tools in soundness and performance. Besides, it identified all vulnerabilities, with a counterexample for each, and proved the fastest approach.

2 TOOL DESCRIPTION

2.1 Tool Overview

Fig. 1 illustrates the architecture of ESBMC-Solidity, where the gray box with solid border represents the new frontend, and the white ones constitute the existing ESBMC's components. The gray box with a dashed border indicates an external element for preprocessing smart contracts: the Solidity compiler. It is used for lexical analysis and parsing, taking a smart contract as input and then transforming it into JSON AST, which is done with the argument *--ast-compact-json*.

The proposed approach takes JSON AST and converts each Solidity AST of its nodes into an equivalent IR one, using the ESBMC's *irept*, a tree-structured IR that preserves a program's semantics. Next, each *irept* node is converted into the corresponding symbol and then added to a table, which is translated into a GOTO program. Then, the latter is processed by the symbolic execution engine (*SymEx*) to generate its static single assignment (SSA) form, which is used to generate verification conditions (VCs) $C \wedge \neg P$, where C represents constraints and P denotes a safety property. Lastly, ESBMC uses off-the-shelf SMT solvers to verify those VCs' satisfiability.

If a property is satisfiable, an execution path leads to a bug in an original Solidity smart contract. Then, when ESBMC detects it, a counterexample is provided, in the form of state traces, to allow its reproduction. It is worth noticing that ESBMC supports several SMT solvers, including Z3 [22], Bitwuzla [23], Boolector [24], MathSAT [25], CVC4 [26], and Yices [27, 28].

2.2 The Grammar-Based Hybrid Conversion Methodology

Given a smart contract as input, the goal of the proposed frontend is to populate the resulting symbol table, where each symbol is represented by the ESBMC's *symbolt* data structure [28]. Furthermore, it shall complete the type-checking procedure of Solidity AST nodes and transform each JSON AST node into its equivalent ESBMC's *irept* one while preserving the associated semantic information. To achieve this goal, we developed this frontend based on the Grammar-Based Hybrid Conversion methodology [29], as an approach specifically devised for that during the development of this work.

Grammar-Based Conversion. The proposed frontend uses the library *nlohmann/json*[1] to process Solidity ASTs in JSON format. When traversing Solidity AST nodes, it uses different functions to transform them into equivalent *irep_t* ones, e.g., *get_var_decl_stmt*, *get_expr*, and *get_statement*, for *variable-declaration-statement*, *expression*, and *statement* nodes, respectively. Besides, each AST node may contain multiple child ones, e.g., the AST node of a *for* loop contains four child nodes: *initialisation*, *condition*, *increment*, and *loop body*. So, during their conversion, the production rules in Solidity grammar documentation are followed [29], so that they are visited in correct order. For instance, the variable-initialisation node of a *for* loop must be visited before the *body* one, as it may be referenced by the latter. If the node for a *body* loop is converted before its variable initialisation, the type checker will fail to handle any reference to it; so, conversion order is guided by production rules.

Hybrid conversion. Three functions must be supported: (1) $assert()$ for defining safety properties; (2) $assume()$ for defining

[1] JSON for Modern C++ - https://github.com/nlohmann/json

constraints; and (3) *nondet*() for assigning non-deterministic values to variables. Consequently, they are implemented by ESBMC as C-style declarations. However, the new frontend works with JSON AST nodes. Besides, since there are more than 70 intrinsic declarations, e.g., forward declarations for nondeterministic types, we instantiate the existing ESBMC's clang frontend to convert those into *irept* nodes to avoid replication, hence generating the symbol table mentioned before. Finally, the latter is further merged with the symbol table generated from the original Solidity AST.

2.3 Illustrative Example

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.26;
3  contract MyContract {
4    uint8 x;
5    uint8 sum;
6    function nondet() public pure
7      returns(uint8) {
8      uint8 i;
9      return i;
10   }
11   function __ESBMC_assume(bool)
12     internal pure { }
13   function func_sat() external {
14     x = 0;
15     uint8 y = nondet();
16     sum = x + y;
17     __ESBMC_assume(y < 255);
18     __ESBMC_assume(y > 220);
19     __ESBMC_assume(y != 224);
20     assert(sum % 16 != 0);
21   }
22 }
```

Figure 2: An example smart contract written in Solidity. In order to instrument the code, the developers need to add the hooks as pure functions in the smart contract. The property is specified using the assert function.

Fig. 2 shows an example of smart-contract verification with ESBMC-Solidity. Indeed, developers can instrument code by adding the hooks, e.g., *nondet* for nondeterministic integers between 0 and 255 and *__ESBMC_assume* for additional constraints. Those help developers narrow down the scope for triggering a bug, hence identifying a set of breaking inputs. Function *func_sat* is the one we need to verify, where two state variables x and *sum* are defined in lines 4 and 5, respectively, while a safety property indicates that $x + y$ should not be a multiple of 16. In addition, constraints are added using *__ESBMC_assume*, in lines 17, 18, and 19, which restrict y as any integer between 220 and 255, but 224. Then, ESBMC will check whether there exists an execution path that satisfies its negation. This way, the verification of *func_sat* becomes a satisfiability problem: given the binary operation expression *"sum = x + y"*, where x is *0* and y is a constrained nondeterministic value, find an execution path where the negation of "$sum\%16! = 0$" is satisfied. ESBMC is then invoked with

```
esbmc <JSON AST> --function func_sat \
    --contract <contract source code> --z3
```

For the smart contract in Fig. 2, ESBMC generates the C and P equations as described in Eq. (1) for constraints and Eq. (2) for property. Eq. (1) shows a conjunction of the constraints represented by assignments. When generating its SSA form, ESBMC uses the temporary variable *temp* to represent the left-hand-side of the safety property specified in line 20, which corresponds to the assignment $temp = sum\%16$ in Eq. (1).

$$C = \begin{bmatrix} y = nondet() \\ \wedge\ sum = y \\ \wedge\ y\,! = 224 \\ \wedge\ temp = sum\%16 \end{bmatrix} \quad (1)$$

$$P = \begin{bmatrix} temp\,! = 0 \end{bmatrix} \quad (2)$$

The resulting VC for satisfiability verification, via SMT solver, is then formed by $C \wedge \neg P$. Consequently, ESBMC reports a property violation and provides a counterexample that contains a trace of states showing the set of assignments and the breaking values that trigger such violation, where y is set with a value 240 as illustrated in Fig. 3.

```
Counterexample:

State 1 file example.sol line 15 function func_sat
----------------------------------------------------
  y = 240

State 2 file example.sol line 16 function func_sat
----------------------------------------------------
  sum = 240

State 6 file example.sol line 20 function func_sat
----------------------------------------------------
Violated property:
  file example.sol line 20 function func_sat
  assertion
  sum % 16 != 0


VERIFICATION FAILED
```

Figure 3: ESBMC-Solidity provides the counterexample, showing the breaking value of y, violating the property "$sum\%16! = 0$".

3 EVALUATION AND BENCHMARKS

ESBMC-Solidity, though an early prototype, that yet covers all production rules, can detect vulnerabilities listed in the smart-contract weakness classification (SWC) registry [31]. So, our evaluation aims to answer three questions.

- EQ1 (**soundness**). Is our approach able to report a confirmed bug in a smart contract?
- EQ2 **(performance)**. Does our approach find a bug in a reasonable amount of time?
- EQ3 **(bug reproduction)**. Is our approach able to provide a counterexample to help reproduce a specific bug?

Table 1: Experimental results, where column "Found" indicates whether a bug was detected, followed by column "CE" showing whether a counterexample was provided. The line "Total Time" represents the CPU time [30] used for verification.

TC	SmartCheck		Slither		Oyente		Mythril		ESBMC-Solidity	
	Found	CE	Found	CE	Found	CE	Found	CE	Found	CE
TC1	No	-	No	-	No	-	Yes	No	Yes	Yes
TC2	No	-	No	-	No	-	Yes	No	Yes	Yes
TC3	No	-	No	-	No	-	Yes	No	Yes	Yes
TC4	No	-	No	-	No	-	Yes	No	Yes	Yes
TC5	Yes	N/A	Yes	N/A	Failed to compile	-	Yes	N/A	Yes	N/A
TC6	No	-	No	-	No	-	Yes	No	Yes	Yes
TC7	No	-	No	-	No	-	Yes	No	Yes	Yes
TC8	No	-	No	-	No	-	Yes	No	Yes	Yes
Total Time	1.160s		0.519s		1.116s		3.106s		0.183s	

3.1 Benchmark Suite Design

A benchmark suite that contains bugs in smart contracts was developed to evaluate ESBMC-Solidity and compare it with other SOTA verification tools [32]. The design of each test case (TC) was guided by the SWC registry [31], as shown in Table 2, while Table 1 shows that all bugs in TCs were detected and confirmed by at least one of the non-ESBMC tools. The test suite and logs are publicly available in Zenodo.[2]

Table 2: Test case design based on SWC registry [31].

SWC Bug ID	Vulnerability	TC
SWC-101	Integer Overflow	TC1,2
	Integer Underflow	TC3,4
SWC-115	Authorization through tx.origin	TC5
SWC-110	Static array out-of-bounds	TC6
	Dynamic array out-of-bounds	TC7,8

3.2 Results

Table 1 shows results for ESBMC-Solidity and other tools. The former found bugs in all TCs confirmed by Mythril, which affirms EQ1. SmartCheck and Slither were able to confirm TC5, which contains a vulnerability reported in the security considerations[3], while detected none for the other TCs. Oyente did not find any bug. Mythril, a tool used in the service MythX^{TM} [21], also reported bugs in all TCs.

Apart from TC5, Mythril and the other non-ESBMC tools failed to provide a counterexample for each TC. However, ESBMC-Solidity did, which affirms EQ3. For TC5, a counterexample is not needed, as a tool should only inform that authorization via *tx.origin* must be avoided. One possible reason for the missing counterexamples could be loss of the original Solidity syntax, as tools either use EVM bytecode, e.g., Oyente and Mythril or rely on various forms of IR that do not preserve the original Solidity declaration references needed for state tracing, e.g., SmartCheck and Slither.

[2]https://doi.org/10.5281/zenodo.5721726
[3]https://docs.soliditylang.org/en/v0.8.6/security-considerations.html

ESBMC-Solidity is the fastest tool, as can be seen in the last line of Table 1, which thus affirms EQ2. Oyente and Mythril work on EVM bytecode and employ simulation for execution path-exploration [33], which might be the reason why they are slower than ESBMC-Solidity. Apart from bug detection, Slither also provides code optimization [19], which might add to the total verification time. To the best of our knowledge, there is no option in Slither to disable the optimization. SmartCheck is implemented in Java, converts Solidity code into an XML-based IR, and uses XPath to query it, while ESBMC-Solidity is implemented purely in C++.

Overall, ESBMC-Solidity presented the best results, reporting bug found in all test cases in 0.183 seconds, faster than other SOTA tools, which answers EQ1 nad EQ2. Regarding the counterexample, ESBMC was the only tool that successfully provided counterexamples for the applicable test cases in our experiment.

4 RELATED WORK

Among the tools we evaluated, Mythril and Oyente use SMT-based symbolic execution to check EVM bytecode and also simulate a virtual machine for execution-path exploration, which might lead to performance degradation [33]. ESBMC-Solidity also uses SMT solvers as backends, but it processes ASTs, so there is no need for environment simulation.

A similar tool that also adopts SMT encoding and solvers to find satisfiability for a property violation is discussed by Alt and Reitwiessner [33]. They developed a component to translate programs into smtlib2 formulae to interface with SMT solvers via their C++ interfaces [33]. The main difference between it and ESBMC-Solidity is that the latter supports code instrumentation, using special functions, which narrows down the scope of inputs that trigger violations. In addition, ESBMC-Solidity can also be extended on top of various existing verification strategies and reasoning techniques provided by ESBMC, such as k-induction [17].

5 CONCLUSIONS

We presented ESBMC-Solidity that checks memory safety and user-defined properties in smart contracts written in the Solidity programming language. We evaluated ESBMC-Solidity against other

SOTA verification tools and overcame them, confirming all presented bugs and providing the associated counterexamples. Other SOTA tools for solidity cannot provide such a counterexample for the violated properties. Although ESBMC-Solidity is an early prototype, it shows promising results. Our current focus is on providing 100% coverage for the language Solidity, including polymorphism, inheritance, special crypto functions, such as Keccak256 and sha256, and multiple returns.

ACKNOWLEDGMENTS

The work in this paper is partially funded by the EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

REFERENCES

[1] Imran Bashir. 2017. *Mastering blockchain.* Packt Publishing Ltd.

[2] Kevin Solorio, Randall Kanna, and Dave H Hoover. 2019. *Hands-on Smart Contract Development with Solidity and Ethereum: From Fundamentals to Deployment.* O'Reilly Media, Incorporated.

[3] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. 2019. Blockchain technology overview. *arXiv preprint arXiv:1906.11078* (2019).

[4] Andreas M Antonopoulos and Gavin Wood. 2018. *Mastering ethereum: building smart contracts and dapps.* O'reilly Media.

[5] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. *Journal of Cases on Information Technology (JCIT)* 21, 1 (2019), 19–32.

[6] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A modular verifier for solidity smart contracts. In *Working Conference on Verified Software: Theories, Tools, and Experiments.* Springer, 161–179.

[7] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. 2021. A survey on formal verification for solidity smart contracts. In *2021 Australasian Computer Science Week Multiconference.* 1–10.

[8] Lucas C Cordeiro, Eddie B de Lima Filho, and Iury V Bessa. 2020. Survey on automated symbolic verification and its application for synthesising cyber-physical systems. *IET Cyber-Physical Systems: Theory & Applications* 5, 1 (2020), 1–24.

[9] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. 2018. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 888–891.

[10] Mikhail R. Gadelha, Rafael S. Menezes, and Lucas C. Cordeiro. 2021. ESBMC 6.1: automated test case generation using bounded model checking. *Int. J. Softw. Tools Technol. Transf.* 23, 6 (2021), 857–861. https://doi.org/10.1007/s10009-020-00571-2

[11] Omar M Alhawi, Herbert Rocha, Mikhail R Gadelha, Lucas C Cordeiro, and Eddie Batista. 2021. Verification and refutation of C programs based on k-induction and invariant inference. *International Journal on Software Tools for Technology Transfer* 23, 2 (2021), 115–135.

[12] Lennon C Chaves, Hussama I Ismail, Iury V Bessa, Lucas C Cordeiro, and Eddie B de Lima Filho. 2019. Verifying fragility in digital systems with uncertainties using DSVerifier v2. 0. *Journal of Systems and Software* 153 (2019), 22–43.

[13] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. 2011. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering* 38, 4 (2011), 957–974.

[14] Jeremy Morse, Lucas C. Cordeiro, Denis A. Nicole, and Bernd Fischer. 2011. Context-Bounded Model Checking of LTL Properties for ANSI-C Software. In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7041)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer, 302–317. https://doi.org/10.1007/978-3-642-24690-6_21

[15] Phillipe A. Pereira, Higo F. Albuquerque, Isabela da Silva, Hendrio Marques, Felipe R. Monteiro, Ricardo Ferreira, and Lucas C. Cordeiro. 2017. SMT-based context-bounded model checking for CUDA programs. *Concurr. Comput. Pract. Exp.* 29, 22 (2017). https://doi.org/10.1002/cpe.3934

[16] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. 2022. Model checking C++ programs. *Softw. Test. Verification Reliab.* 32, 1 (2022). https://doi.org/10.1002/stvr.1793

[17] Mikhail Y. R. Gadelha, Hussama Ibrahim Ismail, and Lucas C. Cordeiro. 2017. Handling loops in bounded model checking of C programs via k-induction. *Int. J. Softw. Tools Technol. Transf.* 19, 1 (2017), 97–114. https://doi.org/10.1007/s10009-015-0407-9

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

[19] 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.

[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] ConsenSys/mythril. v0.22.1. Github. Retrieved Nov 24, 2021 from https://github.com/ConsenSys/mythril

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

[23] Aina Niemetz and Mathias Preiner. 2020. Bitwuzla at the SMT-COMP 2020. *arXiv preprint arXiv:2006.01621* (2020).

[24] Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 174–177.

[25] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. 2008. The mathsat 4 smt solver. In *International Conference on Computer Aided Verification.* Springer, 299–303.

[26] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification.* Springer, 171–177.

[27] Bruno Dutertre. 2014. Yices 2.2. In *International Conference on Computer Aided Verification.* Springer, 737–744.

[28] esbmc/esbmc. v6.8. Github. Retrieved Nov 24, 2021 from https://github.com/esbmc/esbmc

[29] solidity. v0.8.6. soliditylang. Retrieved Nov 24, 2021 from https://docs.soliditylang.org/en/v0.8.6/grammar.html

[30] linux.org. 2019. time(1) — Linux manual page. Retrieved Nov 24, 2021 from https://man7.org/linux/man-pages/man1/time.1.html

[31] SWC Registry. 2020. SmartContractSecurity. Retrieved Nov 24, 2021 from https://swcregistry.io/

[32] Nedas Matulevicius and Lucas C. Cordeiro. 2021. Verifying Security Vulnerabilities for Blockchain-based Smart Contracts. In *XI Brazilian Symposium on Computing Systems Engineering, SBESC 2021, Florianopolis, Brazil, November 22-26, 2021.* IEEE, 1–8. https://doi.org/10.1109/SBESC53686.2021.9628229

[33] Leonardo Alt and Christian Reitwiessner. 2018. SMT-based verification of solidity smart contracts. In *International Symposium on Leveraging Applications of Formal Methods.* Springer, 376–388.