Automated Auditing of Price Gouging TOD Vulnerabilities in Smart Contracts

Sidi Mohamed Beillahi, Eric Keilty, Keerthi Nelaturu, Andreas Veneris, Fan Long
University of Toronto
sm.beillahi@utoronto.ca, eric.keilty@mail.utoronto.ca, keerthi.nelaturu@mail.utoronto.ca,
veneris@eecg.toronto.edu, fanl@cs.toronto.edu

***Abstract*—With the emergence of decentralized finance, smart contracts and their users become more and more susceptible to expensive exploitations. This paper investigates the price gouging transaction order dependency vulnerabilities in smart contracts. A static analysis based approach is proposed to automatically locate and rectify such vulnerabilities, and a prototype tool using Slither, a static analyzer for Solidity, is also developed. All in all, empirical results on a benchmark suite containing 51 Solidity smart contracts show that the proposed methodology can be used successfully to both detect such vulnerabilities and rectify them, or to certify that a Solidity smart contract under question does not contain such vulnerabilities.**

***Index Terms*—Smart contracts, Vulnerabilities, Static analysis**

I. INTRODUCTION

Blockchain is a cryptographically-secure distributed ledger [1], [2]. Blockchain offers an innovative approach that allows establishing trust in an open environment without the need for a centralized authority (or "middle-man") to do so. A smart contract is a piece of program code stored on the blockchain [2] that alters its permanent state. In detail, it is constituted of a set of functions that manipulate this state. Functions can be called either directly by users or indirectly by other smart contracts, through *transactions*. Smart contracts allow performing arbitrarily complex operations (*e.g.*, escrow and insurance) using cryptoassets. An important concept that distinguishes smart contracts from traditional software is the fact that they are immutable, that is, once they are deployed, upgrading them is extremely difficult due to the inherent nature of the blockchain itself.

Although smart contracts have received growing interest in both academia and industry in recent years, the security of smart contracts continues to be an epicenter of discussion. This is because of various exploitations targeting smart contracts that may cause excessive asset losses. For instance, two recent cryptoasset exploitations, namely TheDAO and the Parity wallet bugs, caused a combined loss of $240 million USD. More recently, the fast expansion of decentralized finance (DeFi) applications [3] that use smart contracts is accompanied with many exploitations targeting DeFi smart contracts that caused the additional loss of hundreds of millions USD.

Evidently, applying techniques from formal verification and programming languages to audit smart contracts can help in preventing costly exploitations. In particular, these auditing procedures can provide developers/users with automated tools to locate such vulnerabilities and repair them. Admittedly, such tools can aid in preventing expensive exploitations; they can allow developers to audit their smart contracts before their deployment but also public users to audit potentially malicious smart contracts before they use them. In this paper, we investigate the problems of automated detection and rectification of smart contracts vulnerabilities, namely this of price gouging Transaction Order Dependency (TOD). This vulnerability corresponds to the scenario where the reordering of an honest transaction after an attacker transaction results in changing the final output of the original transaction [4], [5]. Malicious miners can benefit from this by deploying smart contracts that contain price gouging TOD vulnerabilities to exploit public users.

In detail, we propose a static analysis approach to locate and rectify price gouging TOD vulnerabilities. In particular, we present an algorithm that extracts the data dependencies of a smart contract to determine how a change in its state effects the transaction outcome. For example, a currency exchange transaction outcomes depend on an exchange rate that can be manipulated by a concurrent transaction. We implement our algorithm in a prototype tool using `Slither` [6], a static analyzer for Solidity, to extract control and data dependencies of smart contracts. We also evaluate our prototype tool on a benchmark suite of 51 Solidity smart contracts.

In summary, this paper makes the following contributions:

- We study the problem of automated detection and rectification of the price gouging TOD vulnerability.
- We build a prototype implementation of the proposed approach.
- We develop a smart contract benchmark suite of 51 smart contracts to evaluate the proposed methodology. The results show that our approach rectifies the vulnerabilities with just a few changes to the original smart contract.

The rest of the paper is organized as follows. In Section II, we present an overview of price gouging TOD vulnerability. Then in Section II, we describe the technical elements of our proposed approach to automatically locate and rectify this vulnerability. Section IV presents a prototype implemen-

978-1-6654-9538-7/22/$31.00 ©2022 IEEE

tation of our approach for Solidity smart contracts and the empirical results of evaluating the prototype using a smart contract benchmark suite. Finally, we discuss related work and conclusions in Sections V and VI.

II. BACKGROUND AND OVERVIEW

In this section, we illustrate the price gouging TOD vulnerability. We describe a general mechanism to locate and rectify this vulnerability.

A. Price Gouging TOD Vulnerabilities

In price gouging TOD vulnerabilities, the attacker reorders its transaction before an honest pending transaction which results in changing the final output of the honest transaction [5]. In particular, through its transaction the attacker changes the state of the smart contract before the honest transaction is verified. As a motivating example, on the left of Figure 1, we give a market place smart contract written in Solidity programming language [7]. Clients call the function `buy` to purchase an amount of tokens that must be less than the contract inventory, stored in the variable `inventory`. The purchased amount of tokens is computed by dividing the value of `msg.value` by the value of the contract variable `cost`. However, the value of `cost` can be increased by the contract owner, by calling the function `increasePrice`, maliciously while an honest client transaction is pending approval. Therefore, this will result in a loss to the client where the obtained amount of tokens will be affected by the increase cost of a single token. In the remaining paper, for a given price gouging TOD vulnerability we assume that the attacker can only execute a single function, i.e., a setter function, to manipulate the contract's state before the victim's transaction executes. Howver, a smart contract may contain multiple setters functions which the attacker can call to exploit different price gouging TOD vulnerabilities.

B. Locating Price Gouging TOD Vulnerabilities

In this work, we focus on locating price gouging TOD vulnerabilities in smart contracts. Since changing the order between the client transaction and attacker transaction affect the final outcome of the latter, this means that the client transaction outcome is dependent on a state variable that the attacker transaction modifies. Thus, to locate price gouging TOD vulnerabilities we find state variables that effect the outcome of an honest transaction and that can be altered through some setters functions that attacker can call to manipulate the smart contract state. For instance, in the smart contract on the left of Figure 1 the outcome of the transaction calling the function `buy` is affected by the variable `cost` that can be increased by the setter function `increasePrice`.

C. Rectifying Price Gouging TOD Vulnerabilities

A fix to a price gouging TOD vulnerabilty is to add a guard statement to check whether the state of a smart contract is as expected. In particular, this will allow clients to pass values for the states variables that can be altered. Then, in the body of the called function, `require` statements are added to ensure that the current values of the state variables correspond to the expected values passed by the clients. We assume that the clients can query the state variables [1] before issuing its transaction. For instance, on the right of Figure 1, we give the rectified version of the smart contract on the left of Figure 1. Notice that in the final correct version we add an additional parameter to the function `buy`, *i.e.*, `costExpected`, that has the same type as `cost`, *i.e.*, `uint`. Then, in the body of `buy`, we add a `require` statement as a guard to check whether the current value of `cost` corresponds to the passed value of `costExpected`.

Algorithm 1 A procedure for locating price gouging TOD vulnerabilities.

```
procedure ListDependencies($\mathcal{F}, \mathcal{G}$)
    output $\mathcal{Q}$
    $\mathcal{Q} \leftarrow \{\}$
    for each $f \in \mathcal{F}$
        for each $p \in$ outputParams(f)
            $\mathcal{G}' =$ pointToAnalysis(f, p, $\mathcal{G}$)
            for each $x \in \mathcal{G}'$
                if findSetter(x, $\mathcal{F}$) $\neq \perp$
                    $\mathcal{Q}[f] \leftarrow x \uplus \mathcal{Q}[f]$
end procedure
```

III. ANALYSIS APPROACH

Now we present our methodology to automatically locate and rectify price gouging TOD smart contract vulnerabilities.

A. Location Algorithm

Our proposed approach aims to locate the vulnerability in a smart contract and transform the contract's code to rectify the vulnerability without changing the functionality of the contract. We leverage alias and static code analysis to compute relationships between the outcomes of `public` functions that can be called by users and `state variables` that can be manipulated through setter functions. In Algorithm 1, we present our procedure to locate price gouging TOD vulnerability in smart contracts. Given the lists of public functions $\mathcal{F}$ and state variables $\mathcal{G}$ extracted from the `abstract syntax tree (AST)` of a smart contract, the procedure `ListDependencies` computes for each function f in $\mathcal{F}$ the set of state variables $\mathcal{Q}[f] \subset \mathcal{G}$ that the outcome of f depends on and that can be modified by setter functions. In particular, `ListDependencies` computes for each output parameter of f (i.e., `outputParams(f)`) the state variables that it depends on, $\mathcal{G}'$, using the procedure `pointToAnalysis` that leverages existing alias and static code analysis techniques[2] [6], [8] to compute dependency relationships between variables in the context of a given function. For each variable

---

[1] We assume that query functions return the proper values of the state variables. Our approach can be extended to remove this assumption by using static analysis to certify that query functions do not return fraudulent values.

[2] Our proposed algorithm can be complemented with any alias analysis with a reasonable trade-off between precision and performance.

```
contract MMarketPlace {
  address owner;
  uint private cost = 100;
  uint private inventory = 20;

  event Purchase(address _buyer, uint256 _amt);

  function increasePrice(uint increaseCost) {
    require( msg.sender == owner );
    cost += increaseCost;
  }

  function buy() returns(uint) {

    uint amt = msg.value / cost;
    require( inventory > amt );
    inventory -= amt;
    emit Purchase(msg.sender, amt);
  }
}
```

```
contract MMarketPlace {
  address owner;
  uint private cost = 100;
  uint private inventory = 20;

  event Purchase(address _buyer, uint _amt);

  function increasePrice(uint increaseCost) {
    require( msg.sender == owner );
    cost += increaseCost;
  }

  function buy(uint costExpected) returns(uint) {
    require(cost == costExpected);
    uint amt = msg.value / cost;
    require( inventory > amt );
    inventory -= amt;
    emit Purchase(msg.sender, amt);
  }
}
```

Fig. 1: A smart contract with a price gouging TOD vulnerability (left) and its rectified version, without the vulnerability (right).

g in $\mathcal{G}'$, we use the procedure `findSetter` to check whether there exist a public setter function that modifies the value of g. Our proposed algorithm leverages the precision of the above procedures to find the optimal subset of state variables checks for each function. We will call them *dependency variables*.

Once the dependency variables are identified for each public function, our repair mechanism consists of inserting for each dependency variable an input parameter that has the same type in the corresponding public function signature. Following, in the function's body we insert a `require` statement as a guard to check whether the current value of the dependency variable corresponds to the value passed as parameter by the client's transaction that is calling the function. This will allow to check that the state of the dependency variables has not changed since the time when the client issued its transaction.

IV. EMPIRICAL EVALUATION

A. Implementation and Experimental Setup

1) Implementation: We develop a prototype tool implementing the algorithm described in Section III that takes as input a Solidity smart contract. This tool relies on the `Slither` [6] static analyzer framework for Solidity to construct control-flow graphs (CFGs) and dependency relationships in a given Solidity smart contract. Note that in our implementation we consider as public functions, functions with singatures that contain either of the Solidity keywords `public` and `external`. The open-source code for the implementation is available at Github[3] for the interested reader.

2) Experimental Setup: The experiments are run on an Intel Core i3-4170 3.7GHz CPU, 8GB of DDR3 RAM, 256GB SSD machine running Linux Ubuntu 20.04.3LTS operating system in a local network environment.

B. DataSet Collection

For our experiments, we collect a benchmark suite of 51 Solidity smart contracts constituted of three data-sets. The first data-set is constituted of 11 contracts obtained from open-source GitHub repositories. It includes the reference smart contract used in [9] to evaluate static analysis tools for locating TOD vulnerabilities. It also includes two smart contracts extracted from Etherscan [10] that do not have price gouging TOD vulnerabilities to test that the implementation does not flag non-existing price gouging TOD vulnerabilities. The second data-set is constituted of 20 contracts obtained from the benchmark contracts used in [11]. The third data-set is constituted of 20 contracts obtained from the benchmark contracts [12]. The complete dataset can be found on the Github repository with the implementation.

C. Results

We run our prototype tool with the benchmark suite of 51 Solidity smart contracts. In Table I, we report the results of the experiment. The first three columns in Table I list some characteristics of our benchmark suite, i.e., the contract name, the number of lines of code, and the number of functions. The last three columns in Table I list data concerning the application of our tool. The column `nTOD` lists the number of price gouging TOD vulnerabilities our tool locates in each contract. Also, we list the number of lines in contract's code that were altered to rectify these vulnerabilities. We note that the code transformation we apply to smart contracts to rectify the located vulnerabilities is lightweight (column `diff` in Table I). This code transformation, however, does not alter the contracts' behaviors.

The smart contract `BitCash` is the reference contract that was used in [9] to test static analysis tools in locating TOD vulnerabilities. Our tool is able to report the price gouging TOD vulnerability in this contract and rectify it. The two smart contracts `Sale2` and `Crowdsale` do not have price gouging TOD vulnerabilities and we use them to test that our implementation does not give false negatives. The two smart contracts `Sale2-Vulnerable` and `Crowdsale-Vulnerable` are modified versions of `Sale2` and `Crowdsale` contracts, respectively, where we inserted a price gouging TOD vulnerability in each contract.

---

[3] https://github.com/Veneris-Group/TOD-Location-Rectification

TABLE I: Empirical results. Characteristics of contracts: lines of code (`loc`) and number of functions (`nof`). Characteristics after repair: number of repaired vulnerabilites (`nTOD`), lines of code (`loc'`), and lines of code difference (`diff`).

Contract Name	**loc**	**nof**	**nTOD**	**loc'**	**diff**
BitCash	28	2	1	29	2
Sale	71	6	1	72	2
MMarketPlace	21	2	1	22	2
Purchase	31	3	1	32	2
YFT	79	7	2	81	4
TTC	78	7	2	80	4
PrivateSale	40	5	1	43	4
Sale2	125	10	0	125	0
Crowdsale	92	7	0	92	0
Sale2-Vulnerable	129	11	1	130	2
Crowdsale-Vulnerable	96	8	1	97	2
DSTContract	1268	39	9	1278	10
GenesMarket	1262	19	6	1265	3
F3DClick	1926	35	9	1935	9
KnowTokenCrowdSale	228	5	1	229	1
GrowToken	176	14	4	180	4
TrustZen	245	6	8	249	4
GetToken	81	5	2	82	1
Slotthereum	252	21	4	254	2
MyAdvancedToken7	125	12	6	128	3
Crowdsale2	69	10	2	70	1
SaleFix	692	63	2	693	1
Token	144	13	2	145	1
HQ	209	15	4	211	2
Oasis	290	14	7	297	7
SolidStamp	360	20	4	362	2
FairyFarmer	144	22	6	150	6
LISCTrade	399	34	2	400	1
InvestToken	936	92	4	940	4
FoMo3Dshort	1927	78	9	1936	9
DACMI	461	43	7	468	7
Lottery	45	6	1	46	1
kernelFun	118	6	3	121	3
Dickael	270	22	2	274	4
TetherToken	455	11	2	458	3
LinkToken	355	5	1	356	1
TokenSale	61	4	0	61	0
HuanCasino	151	8	1	153	2
MITxSubscriptionPayment	341	2	1	342	1
MultiPadLaunchApp	525	46	3	528	3
TokenUseV2	300	18	6	312	12
Sociol	92	12	1	93	1
Stableupgradeproxy	362	23	3	365	3
GravatarRegistry	67	4	1	68	1
NeoUsd	151	15	2	153	2
GAMCasino	188	13	2	190	2
FabricCrowdSale	105	9	1	106	1
PonziCoin	86	5	3	89	3
CliqStaking	358	28	4	362	4
Betting	225	15	1	226	1
LadaCoin	49	1	1	50	1

D. Limitations and Discussion

In the current setup, our implementation rectifies all detected vulnerabilities; however, it might be the case that some vulnerabilities are not exploitable and repairing them may not be necessary. For instance, this can occur in the case where public users trust a smart contract's owner and they are assured that the contract's state will not be manipulated while their transactions are pending approval.

Another limitation in our implementation is that the static analysis tool `Slither` does not consider inlined assembly statements within the smart contract code. Thus, our implementation might miss dependencies between a transaction's outcome and state variables that can be manipulated.

V. RELATED WORK

Analysis of Smart Contracts. A number of papers have investigated the problem of automated detection of common vulnerabilities in smart contracts. This prior research is either based on symbolic execution engines, *e.g.* [13]–[17], static analysis, *e.g.* [18]–[21], or dynamic analysis, *e.g.*, [22]. The past work based on symbolic execution and dynamic analysis can only establish correctness for *bounded* executions of smart contracts. On the other hand, the works based on static analysis are designed to expose certain coding patterns that are prone to critical vulnerabilities and do not establish full functional correctness. The most closely related work to ours is `Securify` [21] and `Oyente` [15], which investigate TOD among the patterns of vulnerabilities they detect. However, it was shown recently in [9], that those tools may produce false positives and/or false negatives, which is not the case here.

Automated Repairs of Smart Contracts. There is not much work on automated repairs of bugs in smart contracts. In [23], the authors propose an approach to automatically repair four different vulnerabilities in smart contracts, which are intra-function reentrancy, cross-function reentrancy, arithmetic, and `tx.origin` vulnerabilities. However, they do not handle the TOD or the price gouging TOD vulnerabilities we investigate in this paper.

Functional Verification of Smart Contracts. Several previous work has developed frameworks for checking full functional correctness of smart contracts using proof assistants such as `Coq`, `F*`, a nd `Isabelle/HOL` [24]–[28], automated theorem provers (SMT solvers) [29], [30], or predicate abstraction [31]. These works rely on user-provided functional specifications while our work focus on the specific TOD vulnerability pattern, and makes it possible to locate and rectify this vulnerability in smart contracts for which functional specifications do not exist. On the other hand, our work cannot establish the full functional correctness of smart contracts.

VI. CONCLUSION AND FUTURE WORK

An automated technique for detecting and repairing price gouging TOD vulnerability in smart contracts is presented. Using static analysis, we derive dependency relations between public functions that can be called by any user and state variables that can be manipulated by malicious users. We implement our technique in a prototype tool using an existing static analyzer for Solidity. We use the tool to detect and repair price gouging TOD vulnerabilities in 51 Solidity smart contracts demonstrating that it works well in practice. In the future we might extend our work to different kinds of TOD vulnerabilities and other classes of vulnerabilities that are common in smart contracts.

REFERENCES

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system.” 2008. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] V. Buterin, “Ethereum: A next-generation smart contract and decentralized application platform.” 2013. [Online]. Available: https://ethereum.org/en/whitepaper/

[3] “Decentralized finance (defi).” 2021. [Online]. Available: https://ethereum.org/en/defi/

[4] “Swc-114: Transaction order dependence.” 2021. [Online]. Available: https://swcregistry.io/docs/SWC-114

[5] N. Atzei, M. Bartoletti, and T. Cimoli, “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*, ser. Lecture Notes in Computer Science, M. Maffei and M. Ryan, Eds., vol. 10204. Springer, 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[6] J. Feist, G. Grieco, and A. Groce, “Slither: a static analysis framework for smart contracts,” in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 2019, pp. 8–15. [Online]. Available: https://doi.org/10.1109/WETSEB.2019.00008

[7] Solidity, 2021. [Online]. Available: https://docs.soliditylang.org

[8] S. Lagouvardos, N. Grech, I. Tsatiris, and Y. Smaragdakis, “Precise static modeling of ethereum ”memory”,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 190:1–190:26, 2020. [Online]. Available: https://doi.org/10.1145/3428258

[9] A. Ghaleb and K. Pattabiraman, “How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection,” in *ISSTA ’20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 415–427. [Online]. Available: https://doi.org/10.1145/3395363.3397385

[10] Etherscan, 2021. [Online]. Available: https://etherscan.io/

[11] B. Mariano, Y. Chen, Y. Feng, S. K. Lahiri, and I. Dillig, “Demystifying loops in smart contracts,” in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 262–274. [Online]. Available: https://doi.org/10.1145/3324884.3416626

[12] M. Ortner and S. Eskandari, “Smart contract sanctuary.” [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary

[13] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. T. Vechev, “Learning to fuzz from symbolic execution with application to smart contracts,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 531–548. [Online]. Available: https://doi.org/10.1145/3319535.3363230

[14] J. Krupp and C. Rossow, “teether: Gnawing at ethereum to automatically exploit smart contracts,” in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 1317–1333. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[15] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[16] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 653–663. [Online]. Available: https://doi.org/10.1145/3274694.3274743

[17] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts,” in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 664–676. [Online]. Available: https://doi.org/10.1145/3274694.3274737

[18] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, “Madmax: surviving out-of-gas conditions in ethereum smart contracts,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 116:1–116:27, 2018. [Online]. Available: https://doi.org/10.1145/3276486

[19] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “ZEUS: analyzing safety of smart contracts,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[20] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” in *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*, R. Tonelli, G. Destefanis, S. Counsell, and M. Marchesi, Eds. ACM, 2018, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/3194113.3194115

[21] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, “Securify: Practical security analysis of smart contracts,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[22] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, “Online detection of effectively callback free objects with applications to smart contracts,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 48:1–48:28, 2018. [Online]. Available: https://doi.org/10.1145/3158136

[23] T. D. Nguyen, L. H. Pham, and J. Sun, “SGUARD: towards fixing vulnerable smart contracts automatically,” in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1215–1229. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00057

[24] S. Amani, M. Bégel, M. Bortin, and M. Staples, “Towards verifying ethereum smart contract bytecode in isabelle/hol,” in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, J. Andronick and A. P. Felty, Eds. ACM, 2018, pp. 66–77. [Online]. Available: https://doi.org/10.1145/3167084

[25] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Z. Béguelin, “Formal verification of smart contracts: Short paper,” in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, T. C. Murray and D. Stefan, Eds. ACM, 2016, pp. 91–96. [Online]. Available: https://doi.org/10.1145/2993600.2993611

[26] I. Grishchenko, M. Maffei, and C. Schneidewind, “A semantic framework for the security analysis of ethereum smart contracts,” in *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, ser. Lecture Notes in Computer Science, L. Bauer and R. Küsters, Eds., vol. 10804. Springer, 2018, pp. 243–269. [Online]. Available: https://doi.org/10.1007/978-3-319-89722-6_10

[27] Y. Hirai, “Defining the ethereum virtual machine for interactive theorem provers,” in *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds., vol. 10323. Springer, 2017, pp. 520–535. [Online]. Available: https://doi.org/10.1007/978-3-319-70278-0_33

[28] I. Sergey, A. Kumar, and A. Hobor, “Temporal properties of smart contracts,” in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 11247. Springer, 2018, pp. 323–338. [Online]. Available: https://doi.org/10.1007/978-3-030-03427-6_25

[29] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Chakraborty and J. A. Navas, Eds., vol. 12031. Springer, 2019, pp. 161–179. [Online]. Available: https://doi.org/10.1007/978-3-030-41600-3_11

[30] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal verification of workflow policies for smart contracts in azure blockchain," in *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Chakraborty and J. A. Navas, Eds., vol. 12031. Springer, 2019, pp. 87–106. [Online]. Available: https://doi.org/10.1007/978-3-030-41600-3_7

[31] A. Permenev, D. K. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. T. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1661–1677. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00024