SAFEVM: A Safety Verifier for Ethereum Smart Contracts

Elvira Albert, Jesús Correas, Pablo Gordillo
Complutense University of Madrid
Spain

Guillermo Román-Díez
Universidad Politécnica de Madrid
Spain

Albert Rubio
Complutense University of Madrid
Spain

ABSTRACT

Ethereum smart contracts are public, immutable and distributed and, as such, they are prone to vulnerabilities sourcing from programming mistakes of developers. This paper presents SAFEVM, a verification tool for Ethereum smart contracts that makes use of state-of-the-art verification engines for C programs. SAFEVM takes as input an Ethereum smart contract (provided either in Solidity source code, or in compiled EVM bytecode), optionally with assert and require verification annotations, and produces in the output a report with the verification results. Besides general safety annotations, SAFEVM handles the verification of array accesses: it automatically generates SV-COMP verification assertions such that C verification engines can prove safety of array accesses. Our experimental evaluation has been undertaken on all contracts pulled from etherscan.io (more than 24,000) by using as back-end verifiers CPAchecker, SeaHorn and VeryMax.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Software verification and validation**.

KEYWORDS

Smart contracts, Ethereum blockchain, Safety verification.

ACM Reference Format:
Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2019. SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3338999

1 OVERVIEW OF SAFEVM

Each blockchain provides its own programming language to implement smart contracts. Solidity, a Turing complete language, is the most popular language to write smart contracts for the Ethereum platform that are then compiled to EVM (Ethereum Virtual Machine [22]) bytecode. Each instruction executed by the EVM has an associated gas consumption specified by Ethereum. Being security a main concern of Ethereum, the Solidity language contains the verification-oriented functions, assert and require, to check for safety conditions or requirements and terminate the execution if they are not met. As usual, the assert function can be used for verification purposes (e.g., to check invariants), while the require function is used to specify preconditions (e.g., to ensure valid conditions on the inputs or contract state variables, or to validate return values from calls to external contracts). When the Solidity code is compiled into EVM bytecode, the require condition is transformed into a test that checks the condition and invokes a REVERT bytecode if it does not hold. REVERT aborts the whole execution of the smart contract, reverts the state and all remaining gas is refunded to the caller. The assert checks the condition and invokes an INVALID bytecode if it does not hold. When executing INVALID, the state is reverted but no gas is refunded, and hence it has more serious consequences than REVERT: besides the economic consequences of losing the gas, the only information given to the transaction is an out-of-gas error message. The treatment of array accesses is done similarly as for the assert, when an array position is accessed, the generated EVM bytecode checks if the position accessed is within the array bounds and otherwise the INVALID bytecode is executed. Division and related bytecodes like MOD, SMOD, ADDMOD, MULMOD, also lead to executing INVALID when the denominator is zero.

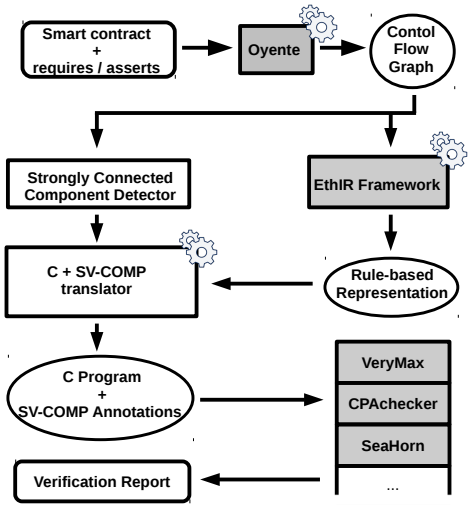

Figure 1: SAFEVM's architecture

Therefore, the INVALID bytecodes are key for the verification of the Ethereum smart contracts, as they capture both assertion violations and several sources of fatal operations (e.g., out-of-bounds access, division by zero). In essence, our approach to the verification of smart contracts consists in decompiling the EVM bytecode for the smart contract into a C program with ERROR annotations (following the SV-COMP format, https://sv-comp.sosy-lab.org/2019/rules.php) to enable their verification using existing tools for the verification of C programs. Developing the verifier from the low-level EVM has

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338999

important advantages: (i) sometimes the source code is not available (e.g., the blockchain only stores the bytecode), (ii) the INVALID bytecodes are visible at the level of bytecode and we can give a uniform treatment to the various safety concerns described above, (iii) our analysis works for any other language that compiles to EVM (e.g., Vyper), and it is not affected by changes in the source language, or by compiler optimizations. Luckily, there are a number of open-source tools that help us in the decompilation process and that we have integrated within our tool-chain.

Fig. 1 depicts the main components of SAFEVM that are as follows (shaded boxes are off-the-shelf used systems not developed by us): (1) *Input.* SAFEVM takes a smart contract, optionally with `assert` and `require` verification annotations. The smart contract can be given in Solidity source code or in EVM compiled code. In the latter case, the annotations have been compiled into bytecode as described above. (2) *CFG.* In either form, the code is given to Oyente [2], a symbolic execution engine that has been extended to compute the complete CFG from the given smart contract. As Oyente does not handle recursive functions, they are already discarded at this step. The CFG generation phase is not described in the paper, we refer to [2, 3]. (3) *EthIR.* The decompilation of the EVM bytecode into a higher-level *rule-based representation* (RBR) is carried out from the generated CFG by EthIR [3]. Technical details of this phase are not described in the paper, we refer to [3]. (4) *C+SV-COMP translator.* We have implemented a translator for the recursive RBR representation into an *abstract* Integer C program (i.e., all data is of type Integer) with verification annotations using the SV-COMP format. Features of the EVM that we cannot handle yet (e.g., bit-wise operations) are abstracted away in the translation (see Sec. 2). INVALID instructions are transformed into ERROR annotations in the C program following the SV-COMP format. (5) *Verification.* Any verification tool for Integer C programs that uses SV-COMP annotations can be used to verify the safety of our C-translated contracts. We have evaluated our approach using three state-of-the-art C verifiers, CPAchecker [6], VeryMax [9], and SeaHorn [14], and the verification report they produce is processed by us to report the results in terms of functions of the smart contract.

Our tool SAFEVM has a very large (potential) user base, as Ethereum is currently the most advanced platform for coding and processing smart contracts. As we will describe in Sec. 3, using SAFEVM we have automatically verified safety of around 20% of *all* functions (depending on the verifier) that might execute INVALID bytecodes from the whole set of contracts pulled from etherscan.io (more than 24,000 contracts), and we have found potential vulnerabilities in functions that could not be verified.

2 TRANSLATION TO C WITH SV-COMP ANNOTATIONS

As motivating example, we use a Solidity contract that implements a lottery system called *SmartBillions* (available at https://smartbillions.com/). We illustrate the safety verification of its internal function `commitDividend` (an excerpt of its code appears to the left of Fig. 2) that commits remaining dividends to the user `wh`. We have shortened the variable names by removing the vowels from the names. Lines marked with ⓘ might lead to executing different sources of INVALID: Line 16 (L16 for short) to a division by zero when `ttlSpply` is 0; at L19 when `lst` ≥ `dvdnds.length` and thus

```
1  contract SmartBillions {
2   struct Wallet {
3       ...,    uint16 lstDvdndPrd;}
4   uint public dvdndPrd;
5   uint[] public dvdnds;
6   mapping(address => uint) blncs;
7   uint public ttlSpply ;
8   mapping (address => Wallet) wllts ;
9
10  function commitDividend(address wh) {
11  ⊕ //require( ttlSpply  > 0);
12  ⊕ //require(dvdndPrd < dvdnds.length);
13    uint lst = wllts[wh].lstDvdndPrd;
14  ⊕ //require(dvdndPrd >= lst );
15    ...
16  ⓘ uint shr=blncs[wh]*0xffffffff/ttlSpply;
17    uint blnc = 0;
18    for(; lst <dvdndPrd;lst++) {
19  ⓘ       blnc += shr * dvdnds[lst ];
20    }
21  ⓘ⊕ assert(lst == dvdndPrd);
22    blnc = (blnc/0xffffffff);
23      ...
24  }
25 }
```

```
block734(s5 ,..., s0,g4,g1,g0,l3,l2) ←
  ...              // block734 instructions
  call (jump734(s7 ,..., s0,g4,g1,g0,l3,l2))
jump734(s7 ,..., s0,g4,g1,g0,l3,l2) ←
  geq(s7,s6),      // lst ≥dvdndPrd
  call (block789(s5 ,..., s0,g4,g0,l3,l2))
jump734(s7 ,..., s0,g4,g1,g0,l3,l2) ←
  lt (s7,s6),      // lst <dvdndPrd
  call (block745(s5 ,..., s0,g4,g1,g0,l3,l2))
block745(s5 ,..., s0,g4,g1,g0,l3,l2) ←
  ...              // block745 instructions
  call (jump745(s9 ,..., s0,g4,g1,g0,l3,l2))
jump745(s9 ,..., s0,g4,g1,g0,l3,l2) ←
  lt (s9,s8),      // lst <dvdnds.length
  call (block759(s7 ,..., s0,g4,g1,g0,l3,l2))
jump745(s9 ,..., s0,g4,g1,g0,l3,l2) ←
  geq(s9,s8),      // lst ≥dvdnds.length
  call (block758(s7 ,..., s0))
block758(s7 ,..., s0) ←
  INVALID
block759(s7 ,..., s0,g4,g1,g0,l3,l2) ←
  // block759 instructions
  ...
  s6 = s7+s6,              // ADD
  s6 = fresh0,             // SLOAD
  s7 = s4,                 // DUP3
  ...
  call (block734(s5 ,..., s0,g4,g1,g0,l3,l2))
```

Figure 2: Solidity code (left) and excerpt of RBR rules of for loop (lines 18-20)

it is accessing a position out of the bounds of the array; and at L21 when the condition within the assert does not hold. In order to be able to verify its safety (i.e., absence of INVALID executions), we add the lines marked with ⊕ that introduce error-handling functions `require` and `assert` in the verification process.

The starting point of our translator is the RBR produced by EthIR [3]. The RBR is composed of a set of rules containing decompiled versions of bytecode instructions (e.g., LOAD and STORE are decompiled into assignments) and whose structure of rule invocations is obtained from the CFG produced by Oyente. The RBR might contain two kinds of rules: sequences of instructions referred to as *blockX*, and conditional jump rules, named *jumpX*, whose first instruction is the Boolean condition used to select between the rules of the function definition. Rule parameters include: the operand stack flattened in variables named s_i, the state of the contract (this is the global data), named g_i, and the local memory (represented by local variables), named l_i. To the right of Fig. 2 we show the fragment of the RBR produced by EthIR for the loop of L18-L20. At rule *block*759 we show the transformation of some EVM bytecodes (the original bytecodes appear in comments `//`) into higher-level RBR instructions. The RBR is already *abstract* in the sense that when variables refer to state or memory locations that are not known they become fresh variables (see variable $\texttt{fresh}_0$ in *block*759) so that a posterior analysis will not assume any value for them (details are in [3]). Observe that the fragment of the RBR contains an INVALID instruction within *block*758 and such block can be executed when $geq(s_9, s_8)$ (see rule *jump*745). By tracking variable assignments, we can infer that s_9 contains the value of `lst` and s_8 the size of `dvdnds`, hence the comparison is checking out-of-bounds array access. The remaining of the section explains the main four phases of the translation from the RBR to an abstract Integer C program.

(1) C functions: Our translation produces, for each non-recursive rule definition in the RBR, a C function without parameters that returns `void`. Recursive rules produced by loops are translated into iterative code. For this part of the translation, we compute the SCC

```
26 int g0 = __VERIFIER_nondet_int();
27  ...
28 int g4 = __VERIFIER_nondet_int();
29 int l0 = __VERIFIER_nondet_int();
30  ...
31 int l3 = __VERIFIER_nondet_int();
32 int who = __VERIFIER_nondet_int();
33 int s0;
34 . . .
35 int s9;
36
37 void block758() {
38   ERROR: __VERIFIER_error();
39 }
40 void block734() {
41  init_loop_0 :
42    // block734   instructions
43    if(s7 >= s6){  // jump734
44      block789();
45      goto end_loop_0; }
46    // block745   instructions
47    if(s9 >= s8){  // jump745
48      block758();
49      goto end_loop_0; }
50   // block759   instructions
51   s6 = s7 + s6
52   s6 = __VERIFIER_nondet_int
          ();
53   s7 = s4;
54    ...
55   goto init_loop_0;
56  end_loop_0:    ;}
```

Figure 3: C translated code with SV-COMP annotations

from the CFG (see Fig. 1) and model the detected loops by means of `goto` instructions. Fig. 3 shows the obtained C functions from the RBR program of Fig. 2. Note that *jump* rules are translated into an *if-then-else* structure.

(2) Types of variables: Solidity basic, signed and unsigned data types are stored into untyped 256-bit words in the EVM bytecode, and the bytecode does not include information about the actual types of the variables. Moreover, most EVM operations do not distinguish among them except for few specific signed operations (SLT, SGT, SIGNEXTEND, SDIV and SMOD). As verifiers behave differently w.r.t. overflow (see details in [6, 9, 14]), our translation allows the user to choose (by means of a flag) if all variables are declared with type `int` in the C program, or of type `unsigned int` with casting to `int` for sign-specific operations. The code in Fig. 3 uses the default `int` transformation. Thus, although in EVM integers have overflow, the interpretation of them as unbounded integers or with overflow will be determined by the available options in the C verification tool (e.g., VeryMax only handles unbounded integers). Besides, instructions that contain `fresh` variables or that are not handled (like SLOAD) are translated into a call to function `__VERIFIER_nondet_int` in order to model the lack of information for them during verification. Observe that function `block734` includes some operations over the different integer variables. Arrays or maps are not visible in the EVM (nor in the RBR). The only information that is trackable about arrays corresponds to their sizes as it is stored in a stack variable that in the C program is stored in an integer variable.

(3) Variable definitions: In order to enable reasoning on them (within their scopes) during verification, SAFEVM translates them in the C program as follows: (i) as we flattened the execution stack, we declare the stack variables as global C variables to make them accessible to all C functions. These variables do not need to be initialized as they take values in the program code; (ii) local variables are defined as global C variables (L29-L31) because a function of the contract might be translated into several C-functions, and all of them need to access the local data. They are initialized at the beginning of the function corresponding to the block in which they are firstly used; (iii) state variables are also translated into global variables accessible by all functions and, as their values when functions are verified are unknown, they are initialized using `__VERIFIER_nondet_int` (L26-L28); and (iv) function input parameters are also defined as global variables (for the same reason as (ii)), whose initial values are not determined (L32).

(4) SV-COMP annotations: The verification of Ethereum smart contracts is done in SAFEVM by guaranteeing the unreachability of the `INVALID` operations in the C-translated code. Following the `SV-COMP` rules, we translate `INVALID` operations into calls to the `__VERIFIER_error` function so that its unreachability can be proven by any verification tool compatible with the `SV-COMP` annotations. An example of an `INVALID` operation can be seen in L38. Verification tools return that the program in Fig. 2 cannot be verified as the `INVALID` instruction could be executed. This is due to the fact that contract state values are unknown, that is: `ttlSpply` is not guaranteed to be different from 0 at L16 and the size of the array `dvdnds` is not guaranteed to be greater than the value of `lst` at L19. Lines L11 and L12 contain the Solidity instructions needed to guarantee that L16 and L19, respectively, will never execute an `INVALID` instruction. The `assert` at L21 can be verified by using the `require` at L14. The inclusion of the `require` annotation also improves the contract as, if it is violated, a `REVERT` rather than an `INVALID` bytecode will be executed, not causing a loss of gas of the transaction (while the gas needed to check it is negligible).

3 EXPERIMENTAL EVALUATION

All components of SAFEVM, except for the C verifiers, are implemented in Python and are open-source. SAFEVM accepts smart contracts written in versions of Solidity up to 0.4.25 and bytecode for the Ethereum Virtual Machine v1.8.18. This section reports the results of our experimental evaluation using SAFEVM with CPAchecker, SeaHorn and VeryMax as verification back-ends. An artifact to try our tool can be downloaded from http://costa.fdi.ucm.es/papers/costa/safevm.ova.

In order to experimentally evaluate SAFEVM, we pulled from etherscan.io all Ethereum contracts whose source code was available on January 2018. This ended up in 10,796 files. From those, we have searched for those files that contain EVM code with `INVALID` instructions, in total 7,323. The first phase of SAFEVM that performs the decompilation into the RBR fails for 1,000 files (this 13.65% is larger but quite aligned with the failing rates of other tools e.g. [1, 8]) and reaches a timeout of 60s for 22 files. Thus, our results are on the remaining 6,301 files, that contain 24,294 contracts with 44,046 public functions that can reach an `INVALID` instruction and 177,549 `INVALID`-free functions. We have tested both the translation to type `int` and `unsigned int` for defining C variables, as mentioned in Sec. 2 for those 44,046 functions. We get the following results by using 60s of timeout (Error denotes an error output by the verifier):

Results	CPAchecker		VeryMax		SeaHorn	
	`int`	`uint`	`int`	`uint`	`int`	`uint`
Verified	19.48%	19.13%	20.32%	20.36%	21.71%	19.57%
Non-Verified	77.04%	79.82%	73.32%	73.44%	77.72%	80.15%
Timeout	3.21%	0.82%	6.29%	6.13%	0.57%	0.28%
Error	0.27%	0.23%	0.07%	0.07%	0%	0%

The results for all verifiers are quite aligned, although VeryMax verifies a slightly lower number of functions, and SeaHorn verifies more functions and less reach a timeout. The interpretation made by the tools regarding the Integer semantics (bounded or unbounded) leads to the only relevant difference in the number of functions verified between both translations.

We have manually inspected, out of the 7,323 files, those files whose addresses start with `0x00` and `0x01` in order to understand

the cases that could not be verified. This is a sample of 29 files (243 public functions) that are available at https://github.com/costa-group/EthIR/tree/master/examples/safevm. The manual inspection on the subset gives 54 false alarms (22.2%), namely: 49 functions were verified by CPAchecker; 140 are correct alarms, most of them produced by asserts introduced by the programmers for safety to abort the execution (e.g. 83 come from `Safemath`); 54 are false alarms (many related to `enum` accesses and other imprecisions in the decompilation phase). More in detail, we have identified four types of situations: (1) false alarms due to *inaccuracy of our tool*: some `assert` statements contain non-integer types (e.g., strings, `enum`, etc.) which cannot be verified as we need a more accurate decompilation (see Sec. 4); (2) correct alarms that require *conditional verification*: some `assert` statements can only be verified for concrete contexts, e.g., we found `asserts` to prevent from under/overflow integer arithmetic operations in a widely used library `SafeMath` that can only be verified for given inputs. In the future we plan to integrate conditional verification [9] to infer the preconditions for the `asserts` to hold; (3) Correct alarms detecting *potential vulnerabilities*: we have detected several `INVALID` operations that could represent a vulnerability in the code (e.g., functions that access an array element without checking the boundary) and we have protected them adding `require` statements that enable subsequent verification; and (4) four functions whose verification results depend on the different semantics used for Integers.

As final observations, we notice that `assert` is overused (contradicting the best practices recommendations of Solidity) and that some contracts can be improved by using `require` to avoid the loss of gas when the `assert` statement does not hold. Finally, we argue that although there is much room for improving the accuracy, the results of our experimental evaluation are very encouraging: we have verified safety w.r.t. `INVALID` bytecodes for around 20% of the functions that might reach `INVALID` fully automatically by using state-of-the-art verifiers.

4 CONCLUSIONS

Verification of Ethereum smart contracts for potential safety and security vulnerabilities is becoming a popular research topic with numerous tools being developed, among them, we have tools based on symbolic execution [13, 15, 17, 18, 20, 21], tools based on SMT solving [16, 19], and tools based on certified programming [5, 7, 12]. There are some tools also that aim at detecting, analyzing and verifying non-functional properties of smart contracts, e.g., those focused on reasoning about the gas consumption [4, 10, 11, 19].

To the best of our knowledge, SAFEVM is the first tool that uses existing verification engines developed for C programs to verify low-level `EVM` code. This opens the door to the applicability of advanced techniques developed for the verification of C programs to the new languages used to code smart contracts. Although our tool is still in a prototypical stage, it provides a proof-of-concept of the transformational approach, and we argue that it constitutes a promising basis to build verification tools for `EVM` smart contracts. Some of the aspects that we aim at improving in future work is the handling of the data stored in the memory, as it is abstracted away by the EthIR component that SAFEVM is using as soon as there are storage operations on memory. Developing a memory analysis for `EVM` smart contracts can be crucial for the accuracy of verification. We also aim at handling bit-wise operations in the future that are extensively used in the `EVM` bytecode. Advanced reasoning for arrays and maps (the only data structures available in Ethereum smart contracts) can be also added to the framework to gain further accuracy. This requires also further work on the decompilation side. Along the same line, learning information on the types of variables during decompilation will have an impact in the accuracy of the verification process.

ACKNOWLEDGMENTS

This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R and MINECO/FEDER, UE project TIN2015-69175-C4-3-R, by Spanish MICINN/FEDER, UE projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, by the CM projects S2018/-TCS-4314 and S2018/TCS-4339, co-funded by EIE Funds of the European Union, and by the UCM CT27/16-CT28/16 grant.

REFERENCES

[1] 2018. Mythril. Available at https://github.com/b-mueller/mythril.
[2] 2018. Oyente: An Analysis Tool for Smart Contracts. https://github.com/melonproject/oyente.
[3] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *ATVA (LNCS)*, Vol. 11138. Springer, 513–520.
[4] E. Albert, P. Gordillo, A. Rubio, and I. Sergey. 2018. GASTAP: A Gas Analyzer for Smart Contracts. *CoRR* abs/1811.10403 (2018). arXiv:1811.10403 http://arxiv.org/abs/1811.10403
[5] S. Amani, M. Bégel, M. Bortin, and M. Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.
[6] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 184–190.
[7] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.
[8] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:1809.03981.
[9] M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. 2015. Compositional Safety Verification with Max-SMT. In *FMCAD*. 33–40.
[10] T. Chen, X. Li, X. Luo, and X. Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE Computer Society, 442–446.
[11] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL* 2, OOPSLA (2018), 116:1–116:27.
[12] I. Grishchenko, M. Maffei, and C. Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST (LNCS)*, Vol. 10804. Springer, 243–269.
[13] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28.
[14] T. Kahsai, J. A. Navas, A. Gurfinkel, and A. Komuravelli. 2015. The SeaHorn Verification Framework. In *CAV*.
[15] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*. The Internet Society.
[16] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena. 2018. Exploiting The Laws of Order in Smart Contracts. *CoRR* abs/1810.11605 (2018). arXiv:1810.11605
[17] J. Krupp and C. Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *USENIX Security Symposium*. USENIX Association, 1317–1333.
[18] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.
[19] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA (LNCS)*, Vol. 11247. Springer, 450–465.
[20] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *ACSAC*. To appear.
[21] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS*. ACM, 67–82.
[22] G. Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger.