



Securing Smart Contract with Runtime Validation

Ao Li
leo@cs.toronto.edu
University of Toronto
Canada

Jemin Andrew Choi
choi@cs.toronto.edu
University of Toronto
Canada

Fan Long
fanl@cs.toronto.edu
University of Toronto
Canada

Abstract

We present SOLYTHESIS, a source to source Solidity compiler which takes a smart contract code and a user specified invariant as the input and produces an instrumented contract that rejects all transactions that violate the invariant. The design of SOLYTHESIS is driven by our observation that the consensus protocol and the storage layer are the primary and the secondary performance bottlenecks of Ethereum, respectively. SOLYTHESIS operates with our novel delta update and delta check techniques to minimize the overhead caused by the instrumented storage access statements. Our experimental results validate our hypothesis that the overhead of runtime validation, which is often too expensive for other domains, is in fact negligible for smart contracts. The CPU overhead of SOLYTHESIS is only 0.1% on average for our 23 benchmark contracts.

CCS Concepts: • Software and its engineering → Runtime environments; • Security and privacy → Domain-specific security and privacy architectures.

Keywords: runtime validation, smart contract, compiler

ACM Reference Format:

Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing Smart Contract with Runtime Validation. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20)*, June 15–20, 2020, London, UK. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3385412.3385982>

1 Introduction

The capability of deploying smart contracts is one of the most important features of many blockchain systems [65]. A smart contract is a program that encodes a set of transaction rules. Once deployed to a blockchain, its encoded rules are

enforced by all participants of the blockchain network, and therefore it eliminates counterparty risks in sophisticated transactions. People have applied smart contracts to a wide range of domains such as finance, supply chain management, and insurance.

Unfortunately, like other programs, smart contracts may contain errors. Errors inside smart contracts are particularly severe because 1) it is often impossible to change a smart contract once deployed; 2) smart contracts often store and manage critical information such as digital assets and identities; 3) errors are treated as intended behavior of the smart contract and faithfully executed by blockchain systems. As a result, errors inside smart contracts often lead to significant financial losses in the real world [21, 27].

To make smart contracts secure and correct, one approach is to build static analysis tools. But such static analysis tools are often inaccurate and generate a large number of false positives and/or false negatives [45, 61]. Another approach is to formally verify the consistency between the implementation and specification of a smart contract [20, 32, 46, 53]. But automated verification tools relying on theorem provers typically have narrow scopes and/or may fail to explore the verification search space, while mechanical verification techniques require human intervention and are often too expensive to apply in practice.

1.1 Runtime Validation with SOLYTHESIS

In this paper, we argue that runtime validation is an effective and efficient approach to secure smart contracts. With access to runtime information, runtime validation techniques can be fully automated and can typically achieve much higher coverage than static analysis techniques. The downside of runtime validation, for traditional programs, is its excessive performance overhead. However, our observation is that *the Proof-of-Work consensus is the primary performance bottleneck of existing blockchain systems*. For example, the consensus protocol of Ethereum can only process up to 38 transactions per second¹, while the execution engine of Parity [10], a popular efficient Ethereum implementation, can process more than 700 transactions on an ordinary laptop with an SSD. Therefore, we hypothesize that *the overhead of runtime validation, which is often too expensive for other domains, is in fact negligible for smart contracts*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '20, June 15–20, 2020, London, UK

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7613-6/20/06...\$15.00
<https://doi.org/10.1145/3385412.3385982>

¹We assume simple payment transactions only to derive this throughput number. In practice, the average throughput of Ethereum is even lower.

To validate our hypothesis, we design and implement SOLYTHESIS, a novel runtime validation tool for Ethereum smart contracts. Unlike static analysis and formal verification techniques that attempt to detect errors in smart contracts offline, SOLYTHESIS works as a source to source Solidity compiler and detects errors at runtime. SOLYTHESIS provides an expressive language that includes quantifiers to allow users to specify critical safety invariants of smart contracts. Taking a potentially insecure smart contract and the specified invariants as inputs, SOLYTHESIS instruments the Solidity code of the smart contract with runtime checks to enforce the invariants. The instrumented contract is guaranteed to nullify all transactions that cause the contract to violate the specified invariants.

The design of SOLYTHESIS is driven by our observation that *the storage layer is the secondary performance bottleneck of Ethereum after the consensus layer*. Our program counter profiling results show that the execution engine of Parity spent over 67% of the running time on components that are relevant to blockchain state load and store operations. Such load and store operations are expensive because 1) it may be amplified to multiple slow disk I/O operations, 2) it is translated by the Solidity compiler into multiple instructions (up to 11 EVM instructions), and 3) the Solidity compiler uses expensive cryptographic hash functions to compute the address of accessed state objects. We therefore design the instrumentation algorithm of SOLYTHESIS to *minimize the number of blockchain state accesses*.

One challenge SOLYTHESIS faces is how to efficiently enforce invariants for ledger-like data structures in smart contracts. Many contracts store critical data into ledger-like structures that map Ethereum addresses to balance-like values. A meaningful invariant for such a ledger will typically include quantifiers to specify constraints for all values in the ledger (e.g., the sum of all balances in a token contract ledger should equal to the total supply of the token). One naive approach to enforce the invariant is to instrument the runtime checks at the end of each transaction, but the runtime checks may have to use loops to access many blockchain state values, which will be extremely expensive.

To address this challenge, SOLYTHESIS instead uses a novel combination of *delta update* and *delta check* techniques. It statically analyzes the source code of each contract function to conservatively determine the set of state values that could be modified and the set of sub-constraints that could be violated during a transaction. It then instruments the instructions to maintain these potentially changed values and to only check these potentially violated constraints. Our results show that these techniques enable SOLYTHESIS to generate secure contracts with negligible overhead on Ethereum. Even if Ethereum or future blockchain systems adopt fast consensus protocols [16, 30, 38, 43, 47, 54, 66], the instrumented contracts will still have an acceptable overhead.

1.2 Experimental Results

We evaluate SOLYTHESIS with 23 smart contracts from ERC20 [5], ERC721 [7], and ERC1202 [6] standards. ERC20 and ERC721 are two Ethereum smart contract standards for fungible and non-fungible tokens. ERC1202 is a draft standard for a voting system which is the key process to many blockchain applications. For each standard, we first compose an invariant and then apply SOLYTHESIS to instrument the smart contracts. Our experimental results show that SOLYTHESIS prevents all vulnerable contracts from violating the defined invariants. The results also validate our hypothesis — the instrumentation overhead is negligible with only 0.1% CPU usage overhead and 3.2KB/s disk write overhead on average. We believe our results encourage future explorations on new languages, new analyses, and new virtual machine designs that can further exploit rigorous runtime validation to secure smart contracts (see Section 7).

Our results also highlight the effectiveness of our instrumentation algorithm. Even if we turn off the consensus layer to simulate the case where the consensus protocol is no longer the performance bottleneck, the SOLYTHESIS instrumentation only causes 24% overhead in transaction throughput on average. In comparison, if we use the naive approach that inserts runtime checks at the end of each transaction, the transaction throughput would be two orders of magnitude smaller. The naive approach can significantly slow down the transaction execution engine, overtaking the consensus layer to become the new performance bottleneck.

1.3 Contributions

This paper makes the following contributions:

- **SOLYTHESIS:** This paper presents SOLYTHESIS, the first source to source runtime validation tool that can enforce global invariants with quantifiers on smart contracts with low overhead.
- **Runtime Validation for Smart Contracts:** This paper presents the first systematic evaluation to show that runtime validation for smart contracts can have negligible overhead. Our results demonstrate that runtime validation is an effective and efficient approach to secure smart contracts.
- **Instrumentation Optimizations:** This paper presents novel delta update and delta check techniques to optimize runtime instrumentation for smart contracts. Our results show that these optimization techniques can improve the transaction processing throughput of the instrumented contracts by two orders of magnitude.

The remaining paper is organized as follows. Section 2 presents our observations on the blockchain system performance bottlenecks and discusses how it would influence the way we secure smart contracts. Section 3 presents a motivating example of SOLYTHESIS. Section 4 and Section 5 present the design and the implementation of SOLYTHESIS,

	ERC20	ERC721	ERC1202
Native	34	11	9
NoConsensus	2181	1184	1439
NoConsensus+Empty	3647	1869	2460

Figure 1. Number of transactions can be processed by Parity client with different configurations.

respectively. We evaluate SOLYTHESIS in Section 6. Section 7 discusses future directions. In Section 8 we discuss related work. We finally conclude in Section 9.

2 Observation

We next present our observations on Ethereum blockchain performance. Ethereum uses a modified version of Nakamoto consensus as its consensus protocol [50, 60, 65]. It stores transactions in a chain of blocks. Participants solve proof-of-work (PoW) problems to generate new blocks to extend the chain.

Consensus Bottleneck: The chain structure of Nakamoto consensus limits the performance of Ethereum — it generates a new block every 13 seconds and the sizes of the blocks are limited by its gas mechanism, which measures the size and the complexity of a transaction [65]. When we ran our experiments, each Ethereum block had a gas limit of 8,000,000 [9]. A simple transaction that only transfers Ether consumes 21,000 gas, and the gas consumption for transactions calling smart contract functions is higher.

To understand the impact of the consensus layer on the overall performance of Ethereum, we run experiments with representative transaction traces for the following contracts: the BEC contract [2], an ERC-20 smart contract for managing fungible BEC tokens, the DozDoll contract [4], an ERC-721 smart contract for managing non-fungible DOZ tokens, and an example contract in the ERC-1202 contract standard [6], which is designed for hosting voting on Ethereum. We use Parity, the most efficient Ethereum client that is publicly available, to start a private Ethereum network to process these transactions. All experiments were performed on an AWS EC2 m5.xlarge virtual machine, which has 4 cores and 16GB RAM.

Figure 1 presents the number of transactions that can be processed per second for each contract. Note that we run our experiments with three different configurations: 1) we use the Ethereum state at the block height 5,052,259 as the initial state and run PoW consensus to pack and process transactions (corresponding to the group “Native” in Figure 1); 2) we remove the PoW consensus limit so that Parity can process as many transactions as its transaction execution engine allows (corresponding to the group “NoConsensus” in Figure 1); 3) we remove the PoW limit and start Parity with an empty genesis state instead (corresponding to the group “NoConsensus+Empty” in Figure 1).

	Storage	Verifier	EVM	Other
ERC20	67.0%	25.9%	3.9%	3.2%
ERC721	73.5%	18.3%	5.7%	2.5%
ERC1202	73.1%	20.5%	3.6%	2.8%

Figure 2. The performance counter results for different components of Parity client.

The results in Figure 1 show that Parity only processes 9 to 34 transactions per second for the ERC-20, ERC-721, and ERC-1202 contracts. If the consensus protocol is disabled, Parity processes 1184 to 2181 transactions per second for the same set of contracts. We therefore have the following observation and hypothesis:

Observation 1: The consensus protocol is the primary performance bottleneck of Ethereum clients.

Hypothesis: The overhead of runtime validation can be negligible for smart contracts, because the transaction execution engine is not the primary bottleneck.

Storage Bottleneck: The results in Figure 1 show that Parity would run much faster with an empty initial state than with an initial state corresponding to the real Ethereum network at block 5,052,259. This is because all Ethereum clients, including Parity, store the blockchain state as a Merkle Patricia Tree (MPT) [65] on the disk. Each update on the blockchain state will be amplified to multiple disk I/O operations depending on the height of the MPT. When Parity starts with an empty state, the MPT is simpler. Therefore, there will be fewer I/O operations than starting with a complicated state.

To better understand the performance impact of the blockchain state updates (i.e., load/store EVM instructions), we profile Parity in our experimental runs to collect the performance counters of different components in Parity. Figure 2 presents the profiling results. It classifies the performance counters into four categories: 1) the modules for the blockchain state updates, including the RocksDB storage layer and the functions in the EVM interpreter for the load/store Solidity statements; 2) the modules for verifying signatures in transactions; 3) the modules for other EVM instructions except for loads/stores; 4) all remaining modules.

Our results show that the blockchain state updates account for more than 67% of the performance counters for all experimental runs of Parity when we turn off the consensus layer. The load and store operations to the blockchain state are particularly expensive because 1) these operations could trigger one or more disk I/O operations, 2) the Solidity compiler often generates expensive SHA3 EVM instructions to compute the address operands of these operations [42, 53], and 3) the Solidity compiler often translates one state load/store statement into multiple EVM instructions (up to eleven). We

```

1  contract ERC1202Example {
2    mapping(uint => uint[]) options;
3    mapping(uint => bool) internal isOpen;
4    mapping(uint => mapping(address => uint256)) public
   ↪ weights;
5    mapping(uint => mapping(uint => uint256)) public
   ↪ weightedVoteCounts;
6    mapping(uint => mapping(address => uint)) public
   ↪ ballots;
7    function createIssue(...) public { ... }
8    function winningOption(...) public returns (uint
   ↪ option) { ... }
9    function vote(uint issueId, uint option) public
   ↪ returns (bool success) {
10     require(isOpen[issueId]);
11     uint256 weight = weights[issueId][msg.sender];
12     // weightedVoteCounts[issueId]
   ↪ [ballots[issueId][msg.sender]] -= weight;
13     weightedVoteCounts[issueId][option] += weight;
14     // assert(weightedVoteCounts[issueId][option] >=
   ↪ weight);
15     ballots[issueId][msg.sender] = option;
16     emit OnVote(issueId, msg.sender, option);
17     return true;
18   }
19 }

```

Figure 3. Simplified source code from a voting contract.

therefore have the following observation and the implied design decision:

Observation 2: The state storage layer is the secondary performance bottleneck of Ethereum clients.
Design Decision: Runtime validation tools should minimize the number of instrumented load/store statements that access blockchain state.

3 Example

We next present a motivating example to illustrate the design of SOLYTHESIS. Figure 3 shows a simplified source code from the draft of ERC1202 [6], which is a technical standard draft that defines a set of function interfaces to implement a voting system. A voting system allows a user to vote on different issues and returns the winning option. This example is used for illustration purposes in the ERC1202 draft, but it contains a logic error.

In Figure 3, the `vote()` function (lines 9-18) updates the vote of a transaction initiator given an option and an issue. The contract implements `vote()` and other functions with five state variables. `options` (line 2) stores the available options of each issue; `isOpen` (line 3) stores the current status of the issue; `weights` (line 4) stores the weight of each voter on each issue; `weightedVoteCount` (line 5) stores the total weighted count of each option on each issue; `ballots` (line 6) stores the vote of each voter on each issue.

```

1  s = Map (a, c) Sum weights[a][b] Over (b) Where
   ↪ ballots[a][b] == c;
2  ForAll (x, y) Assert y == 0 || s[x][y] ==
   ↪ weightedVoteCounts[x][y];

```

Figure 4. Invariant for ERC1202.

In the implementation of the `vote()` function, it first fetches the weight of the transaction initiator and updates the weighted votes of the given option and issue (lines 11 and 13 in Figure 3). However, the implementation contains two errors: 1) the original implementation fails to consider the case where the transaction initiator votes multiple times on the same issue; 2) an attacker may trigger an overflow error at line 13 to illegally modify the weighted vote count. Note the original implementation misses line 12 and line 14 which are necessary to fix these errors. We next apply SOLYTHESIS to the contract and describe how SOLYTHESIS instruments the contract to nullify these errors.

Specify Invariant: Note that both of these errors cause the contract to potentially violate the ERC-1202 invariant where the total weighted count of an option on an issue should equal to the sum of all weights of voters who voted for the option. To apply SOLYTHESIS, we first specify this invariant shown as Figure 4. The first line in Figure 4 defines an intermediate map `s` that corresponds to the sum of all weights of the voters that voted for issue and option pairs. The free variable `a` iterates over issues; `b` iterates over voters; `c` iterates over options of an issue. For each map entry `s[a][c]`, it iterates over all voters to sum the weights of those voters who voted for the option `c` on the issue `a`. The second line defines a constraint with the `ForAll` quantifier to specify that for all pairs of issues and options, the intermediate map `s` should equal to the calculated `weightedVoteCount` in the state. Note that in ERC-1202, zero is a special option id to denote that a voter did not vote yet. Therefore our invariant excludes the option id zero.

Figure 4 highlights the expressiveness of the invariant language in SOLYTHESIS. A user can refer to any state variable in the contract (e.g., `weights`, `ballot`, and `weightedVoteCount` in Figure 4), define intermediate values including maps, and specify constraints on state variables and defined intermediate values. SOLYTHESIS supports sophisticated operations such as a conditional sum over values inside maps (e.g., line 1 in Figure 4) and a `ForAll` quantifier to define a group of constraints for multiple state values at once (e.g., line 2 in Figure 4).

Instrument the Contract: One naive approach to enforce the invariant in Figure 4 is to instrument a brute force check at the end of every transaction. This would cause prohibitive overhead because of the iterative sum operation and the quantifier constraint. It would cost two or even three nested loops to check the invariant for every transaction.

```

1  contract ERC1202Example {
2      mapping (uint => mapping(uint => uint)) s;
3      uint256[] x_arr;
4      uint256[] y_arr;
5      ...
6      function vote (uint issueId, uint option) public
7      ↪ returns (bool success) {
8          require(isOpen[issueId]);
9          uint256 weight = weights[issueId][msg.sender];
10
11         x_arr.push(issueId);
12         y_arr.push(option);
13         weightedVoteCounts[issueId][option] += weight;
14
15         x_arr.push(issueId);
16         y_arr.push(ballots[issueId][msg.sender]);
17         assert(s[issueId] [ballots[issueId][msg.sender]] >=
18             ↪ weights[issueId][msg.sender]);
19         s[issueId] [ballots[issueId][msg.sender]] -=
20             ↪ weights[issueId][msg.sender];
21         ballots[issueId][msg.sender] = option;
22         x_arr.push(issueId);
23         y_arr.push(ballots[issueId][msg.sender]);
24         s[issueId] [ballots[issueId][msg.sender]] +=
25             ↪ weights[issueId][msg.sender];
26         assert(s[issueId] [ballots[issueId][msg.sender]] >=
27             ↪ weights[issueId][msg.sender]);
28
29         emit OnVote(issueId, msg.sender, option);
30         for (uint256 index = 0; index < x_arr.length; index
31             ↪ += 1)
32             assert(y_arr[index] == 0 || s[
33                 ↪ x_arr[index]][y_arr[index]] ==
34                 ↪ weightedVoteCounts[x_arr[index]]
35                 ↪ [y_arr[index]]);
36         x_arr.length = 0;
37         y_arr.length = 0;
38         return true;
39     }
40     ...
41 }

```

Figure 5. Instrumented voting contract.

SOLYTHESIS instead instruments code to perform *delta updates* and *delta checks* to reduce the overhead. The intuition is to maintain intermediate values such as s incrementally and to instrument updates and checks only when necessary. Figure 5 presents the instrumented contract code generated by SOLYTHESIS for our example.

Instrument Delta Updates: For every intermediate value such as s in Figure 4, SOLYTHESIS instruments the contract to add it as a state variable. Then for every write operation to an original state variable in the contract (e.g., line 18 in Figure 5), SOLYTHESIS performs static analysis to determine whether modifying the original variable may cause the intermediate value to change. If so, SOLYTHESIS instruments code to update the intermediate value conditionally (e.g., lines 17 and 21 in Figure 5).

Compute Free Variable Bindings: For every state write operation in the contract, SOLYTHESIS computes *free variable bindings* against each rule in the invariant. For example, for the write operation to ballots at line 18 in Figure 5, SOLYTHESIS determines that the operation may change the intermediate value defined in the first line of the invariant. SOLYTHESIS also binds the free variable a in the invariant to $issueId$, binds b to $msg.sender$, and binds c to $ballot[issueId][msg.sender]$. These bindings indicate that the write to $ballot[issueId][msg.sender]$ may only change the intermediate value $s[issueId][ballot[issueId][msg.sender]]$. Therefore SOLYTHESIS instruments code at lines 17 and 21 to only update this value.

Instrument Delta Check: For every state write operation, SOLYTHESIS also checks it against Assert constraints in the invariant. For example, for the write operation to ballots at line 18 in Figure 5, SOLYTHESIS first runs its binding analysis against the constraint at line 2 in Figure 4 which contains a ForAll quantifier. This analysis determines that the write operation may cause the contract to violate the constraint when x binds to $issueId$ and y binds to $ballot[issueId][msg.sender]$.

SOLYTHESIS defines additional arrays like x_arr and y_arr to collect such free variable combinations that may lead to constraint violations lines 3 and 4 in Figure 5. SOLYTHESIS then instruments statements at lines 14-15 and 19-20 to appropriately maintain these arrays. SOLYTHESIS finally instruments a loop at the end of the transactions to only check these potentially violated constraints (lines 25-26). Note that SOLYTHESIS further optimizes the instrumentation to use global volatile memory to hold x_arr and y_arr to avoid blockchain state access (see Section 5.2). We omit this implementation detail in Figure 5 for brevity.

Nullify Errors: SOLYTHESIS generates the instrumented program as its output shown as Figure 5. This instrumented program will enforce the invariant faithfully during runtime and nullify any malicious transactions that cause the contract to violate the invariant. In our example, we deploy the instrumented contract to Ethereum and intentionally trigger the error by sending transactions to vote for an issue multiple times. The instrumented assertion at line 22 catches this error and aborts the offending transactions. Therefore SOLYTHESIS successfully nullifies the error.

4 Design

We next formally present the design of SOLYTHESIS. In this section, we use the notation $s[X/Y]$ to denote replacing every occurrence of X in the statement s with Y . To avoid confusion, we will use “ $[[]$ ” instead of “[]” to denote the indexing of map variables. We also use the notation \vec{x} to denote a list of variables x_1, x_2, \dots

$$\begin{aligned}
& \text{const} \in \text{Int} & x, y \in \text{FreeVar} & v \in \text{Var} \\
& e, e_1, e_2 \in \text{Expr} := \text{const} \mid v \mid v[x_1][x_2][\dots] \mid e_1 \text{ aop } e_2 \\
& c, c_1, c_2 \in \text{CondExpr} := e_1 \text{ cop } e_2 \mid e_1 == x \mid c_1 \wedge c_2 \\
& r \in \text{Rule} := v = \text{Map } x_1, x_2, \dots \text{ Sum } e \\
& \quad \text{Over } y_1, y_2, \dots \text{ Where } c; \mid \\
& \quad \text{ForAll } x_1, x_2, \dots \text{ Assert } c; \mid r_1 r_2
\end{aligned}$$

Figure 6. The invariant specification language.

4.1 Invariant and Contract Languages

Invariant Language: Figure 6 presents the syntax of our invariant specification language with integers, variables, arithmetic expressions, conditional expressions, intermediate value declarations, and constraints. There are three types of variables: state variables, intermediate variables, and free variables. State variables are variables declared in smart contracts and stored in persistent storage. Intermediate variables correspond to the intermediate values defined in the invariant rules of the form “ $v = \text{Map} \dots$ ”. Note that in our language we do not distinguish these two kinds of variables, because SOLYTHESIS instruments code to declare intermediate variables as state variables. Free variables are only used together with quantifiers in Map and ForAll rules. They act as indexes for a defined intermediate map, as iterators for a sum operation, or as quantifier variables to define a group of constraints at once.

Expressions are built out of variables and integer constants. “aop” represents an arbitrary binary arithmetic operator; “cop” represents an arbitrary integer comparison operator. There are three possible types for expressions, integers for scalar values, maps that have integer keys, and booleans for conditional expressions. All variables and expressions should be well-typed. $e[x]$ denotes accessing the map e at the index x , where e must be an expression with a map type and x must be a free variable with the integer type. It is possible to have multi-dimensional maps. In the rest of this section, the notation $e[\vec{x}]$ is an abbreviation of $e[x_1][x_2][\dots]$ for accessing such multi-dimensional maps.

The language has two kinds of rules: intermediate value declaration rules and constraint declaration rules. An intermediate value declaration can define a map value indexing over a list of free variables and can conditionally and iteratively sum over state variable values to compute each map entry. For a rule of the form “ $v = \text{Map } \vec{x} \text{ Sum } e \text{ Over } \vec{y} \text{ Where } c$ ”, each entry of the intermediate map v is defined as follows:

$$\forall \vec{x} \ v[\vec{x}] = \sum_{\vec{y}} \begin{cases} e(\vec{x}, \vec{y}) & \text{if } c(\vec{x}, \vec{y}) \text{ is True} \\ 0 & \text{otherwise} \end{cases}$$

A constraint declaration rule of the form “ForAll \vec{x} Assert c ” checks all possible assignments of \vec{x} to ensure that $c(\vec{x})$ is

$$\begin{aligned}
& c \in \text{Int} & v, v_1 \in \text{StateVar} & t, t_1 \in \text{TempVar} \\
& a, a_1 \in \text{Address} := v \mid v[e_1][e_2][\dots] \\
& e, e_1, e_2 \in \text{Exp} := c \mid t \mid e_1 \text{ op } e_2 \\
& s, s_1, s_2 \in \text{Statement} := t = e; \mid s_1 s_2 \mid t = \text{load } a; \mid \\
& \quad \text{store } a, e; \mid \text{if } e \{s_1\} \mid \\
& \quad \text{for } t_1, t_2, \dots \text{ in } v \{s_1\} \mid \\
& \quad \text{assert } e;
\end{aligned}$$

Figure 7. The core language for smart contracts.

always satisfied. Note that free variable lists after Map and ForAll can be empty. Therefore, users can use these rules to define scalar values and simple constraints as well. Also note that free variables in invariants iterate over all defined keys in maps where those variables are used as indexes. For example, in “ForAll \vec{x} Assert $m[x] > 0$ ”, x should iterate over values that correspond to defined keys in m .

Smart Contract Language: Figure 7 presents the core language of smart contracts that we use to illustrate SOLYTHESIS. “op” denotes an arbitrary binary operator. We do not distinguish normal expressions and conditional expressions in our contract language. There are two kinds of variables: state variables that may be referred in the invariant and temporary scalar variables that are local to the contract program.

“load” and “store” are statements for accessing blockchain state variables. Similar to Solidity, state variables can be either scalar values or maps. “for t_1, t_2, \dots in $v \{s_1\}$ ” would iterate over all possible assignment combinations of t_1, t_2, \dots based on how t_1, t_2, \dots are used as indexing variables for the map v in s_1 . If any of these variables are not used as indexing variables for v , this statement is undefined. Loop statements in our language capture the most common usage of loops in Solidity contracts, and its syntax simplifies the presentation of our instrumentation algorithm. Note that in Solidity, every state variable has to be declared before its use. We omit the declaration syntax for brevity.

4.2 Instrumentation Algorithm

Figure 8 presents the SOLYTHESIS instrumentation algorithm. Given a program P as a list of contract statements and an invariant R as a list of rules, the algorithm produces an instrumented program P' that enforces the invariant dynamically. The algorithm has two parts. Lines 2-12 handle the intermediate value declarations in R , while lines 13-24 handle the quantifier constraint rules in R .

For every defined intermediate value v , the algorithm instruments a fresh state variable declaration for the value (line 4). The algorithm then inspects every store statement s in P and instruments the contract to maintain v (lines 5-12). The algorithm first computes possible bindings of free variables in the definition of v (lines 6-7). A binding is a set

Input : Program P as a list of statements and a list of invariant rules R

Output: The instrumented program as a list of statements

```

1  $P' \leftarrow P$ 
2 for  $r \in R$  do
3   if  $r = \text{"}v = \text{Map } \vec{x} \text{ Sum } e \text{ Over } \vec{y} \text{ Where } c;\text{"}$ 
4     then
5       Assume  $v$  is fresh. Declare  $v$  in  $P'$ .
6       for  $s = \text{"store } a, _;\text{"} \in P$  do
7          $\mathcal{B} \leftarrow \text{BindExpr}(a, e) \cup \text{BindExpr}(a, c)$ 
8          $b \leftarrow \text{BindCond}(a, c)$ 
9          $pre \leftarrow \text{"if } c \{ t = \text{load } v[\vec{x}];$ 
10         $t' = t - e; \text{store } v[\vec{x}], t'; \}$ "
11         $pre \leftarrow \text{Rewrite}(pre, \mathcal{B}, b)$ 
12         $post \leftarrow \text{"if } c \{ t = \text{load } v[\vec{x}];$ 
13         $t' = t + e; \text{store } v[\vec{x}], t'; \}$ "
14         $post \leftarrow \text{Rewrite}(post, \mathcal{B}, b)$ 
15        Insert  $pre$  before  $s$  and  $post$  after  $s$  into  $P'$ 
16     else if  $r = \text{"ForAll } \vec{x} \text{ Assert } c;\text{"}$  then
17       Declare a fresh map  $\alpha$  in  $P'$  corresponding to  $r$ 
18       for  $s = \text{"store } a, _;\text{"} \in P$  do
19          $\mathcal{B} \leftarrow \text{BindExpr}(a, c)$ 
20          $pre \leftarrow \alpha[\vec{x}] = 1$ 
21          $pre \leftarrow \text{Rewrite}(pre, \mathcal{B}, \langle \perp, \perp \rangle)$ 
22         Insert  $pre$  before  $s$  in  $P'$ 
23      $P \leftarrow P'$ 
24 for  $r = \text{"ForAll } \vec{x} \text{ Assert } c;\text{"} \in R$  do
25    $\alpha \leftarrow$  The defined map that corresponds to  $r$ 
26    $s' \leftarrow \text{"for } \vec{x} \text{ in } \alpha \{ \text{assert } c; \}$ 
27   Insert  $s'$  at the end of  $P'$ 
28 return  $P'$ 

```

Figure 8. Instrumentation algorithm.

of pairs of free variables in the definition and expressions in the contract. The binding corresponds to possible entries of v (if v is a map) that s may influence via state variable write operations. The algorithm prepares statement templates for updating v (lines 8 and 10), rewrites free variables in these templates based on the computed bindings (lines 9 and 11), and then instruments the rewritten statements into P' (line 12). The update strategy is to first subtract the old expression value (e.g., pre in lines 8-9) before the execution of s , and then add the new expression value (e.g., $post$ in lines 10-11) after the execution of s . See Section 4.3 for our binding and rewrite algorithms.

For every quantifier constraint rule r , the algorithm also instruments the declaration for a fresh state map variable

$$\begin{array}{c}
\frac{const \in \text{Int}}{\text{BindExpr}(a, const) = \emptyset} \qquad \frac{a = v}{\text{BindExpr}(a, v) = \{\emptyset\}} \\
\frac{a = v' \quad v' \neq v}{\text{BindExpr}(a, v) = \emptyset} \qquad \frac{a = v'[\dots] \quad e = v[\dots] \quad v' \neq v}{\text{BindExpr}(a, e) = \emptyset} \\
\frac{a = v[x_1][\dots][x_k] \quad e = v[e_1][\dots][e_k]}{\text{BindExpr}(a, e) = \{\langle x_1, e_1 \rangle, \dots, \langle x_k, e_k \rangle\}} \\
\frac{e = e_1 \text{ aop } e_2}{\text{BindExpr}(a, e) = \text{BindExpr}(a, e_1) \cup \text{BindExpr}(a, e_2)} \\
\frac{c = e_1 \text{ cop } e_2}{\text{BindExpr}(a, c) = \text{BindExpr}(a, e_1) \cup \text{BindExpr}(a, e_2) \quad \text{BindCond}(a, c) = \emptyset} \\
\frac{c = \text{"}e_1 == x\text{"} \quad e_1 \text{ contains } x}{\text{BindExpr}(a, c) = \text{BindExpr}(a, e_1) \quad \text{BindCond}(a, c) = \langle \perp, \perp \rangle} \\
\frac{c = \text{"}e_1 == x\text{"} \quad e_1 \text{ does not contain } x}{\text{BindExpr}(a, c) = \text{BindExpr}(a, e_1) \quad \text{BindCond}(a, c) = \langle x, e_1 \rangle} \\
\frac{c = \text{"}c_1 \wedge c_2\text{"} \quad \text{BindCond}(a, c_1) \neq \langle \perp, \perp \rangle}{\text{BindExpr}(a, c) = \text{BindExpr}(a, c_1) \cup \text{BindExpr}(a, c_2) \quad \text{BindCond}(a, c) = \text{BindCond}(a, c_1)} \\
\frac{c = \text{"}c_1 \wedge c_2\text{"} \quad \text{BindCond}(a, c_1) = \langle \perp, \perp \rangle}{\text{BindExpr}(a, c) = \text{BindExpr}(a, c_1) \cup \text{BindExpr}(a, c_2) \quad \text{BindCond}(a, c) = \text{BindCond}(a, c_2)}
\end{array}$$

Figure 9. Binding algorithm.

α . Note that because of the ForAll quantifier, r may correspond to multiple constraint instances. To handle this, the algorithm inspects every store statement s in P and uses its binding algorithm to determine whether the execution of s may cause some previously satisfied constraint instance of r to be violated again. If so, the algorithm sets the corresponding entry in α (lines 16-19) to mark these instances. The algorithm finally instruments a for loop iterating over α at the end of the contract to check these potentially violated constraints (lines 21-24).

4.3 Binding and Rewrite Algorithms

Free Variable Binding: Figure 9 presents our binding algorithm. It defines two functions $\text{BindExpr}()$ and $\text{BindCond}()$ that are used in our instrumentation algorithm (see Figure 8). Given a modified address a in a contract store statement and an expression e in an invariant rule, $\text{BindExpr}(a, e)$ returns a set of binding maps that maps free variables in e to indexing expressions in a . To compute $\text{BindExpr}(a, e)$, SOLYTHESIS recursively traverses the structure of e and looks for the map

indexing expressions that match the state variable in a . The fifth rule in Figure 9 creates a binding map for such matching index expressions for the matching free variables.

Note that because e is an expression from the invariant, it can have multiple instantiations with different free variable assignments. Intuitively, a binding map corresponds to possible free variable assignments for e that the state value at the address a may influence. Because one free variable may map to multiple indexing expressions in a binding map, in our notation we represent the binding map as a set of pairs of free variables and indexing expressions. For example, $\text{BindExpr}(a, e) = \{\langle x_1, e_1 \rangle, \langle x_2, e_2 \rangle\}$ means that state value changes at the address a may influence the evaluation of the instantiations of e in which we replace x_1 with e_1 and x_2 with e_2 . If $\text{BindExpr}(a, e)$ returns a set that contains multiple binding maps, it means that the state value changes at a may influence instantiations that are represented by all of these maps.

Given a modified address a and a condition expression c in an invariant rule, $\text{BindCond}(a, c)$ returns a tuple pair of a free variable and an expression. The instrumentation algorithm in Figure 8 uses $\text{BindCond}()$ to handle intermediate value declaration rules only. The computation of $\text{BindCond}()$ scans for the condition of the form $e_1 == x$, where x is a free variable. If an intermediate value declaration rule has such a condition, SOLYTHESIS can directly infer that the free variable x must equal to the expression e_1 for all cases.

Free Variable Rewrite: Figure 10 presents the definition of $\text{Rewrite}()$. Given a statement template s that may contain free variables, a set of binding maps \mathcal{B} extracted from expressions with $\text{BindExpr}()$, and a binding tuple b extracted from conditions with $\text{BindCond}()$, $\text{Rewrite}(s, \mathcal{B}, b)$ produces a new statement with all free variables being rewritten based on the provided bindings.

The Rewrite algorithm iterates over binding maps in \mathcal{B} and generates one statement instantiation based on each binding map. For each free variable in s , it detects whether it has exactly one appearance in the binding map. If the free variable appears multiple times, i.e., the free variable maps to multiple indexing expressions, the algorithm instruments additional if statement guards to ensure that all of the matched indexing expressions have the same value (line 7). The algorithm then removes redundant tuples and only keeps one of these indexing expressions (line 8).

If the free variable does not appear in the binding map, the algorithm would conservatively wrap the statement with a for loop to handle this unbounded free variable and add a tuple that maps the unbounded variable to the iterator variable of the loop (lines 10-13). Such a scenario is rare in practice because it often corresponds to awkward less-common constraints for smart contracts. At line 14, the binding map B' should map each free variable in s exactly to each expression. The algorithm then replaces these free variables with their corresponding expressions (lines 14-18).

Input : The original statement s , a set of binding maps \mathcal{B} extracted from expressions, and a binding b extracted from conditions.

Output: The rewritten statement respecting the bindings.

```

1  $s' \leftarrow \emptyset$ 
2 for  $B \in \mathcal{B}$  do
3    $s'' \leftarrow s$ 
4    $B' \leftarrow B \cup \{b\}$ 
5   for  $x \in \text{FreeVar}$ , where  $x$  appears in  $s$  do
6     if  $x$  appears in  $B'$  multiple times as
7        $\langle x, e_1 \rangle, \dots, \langle x, e_k \rangle$  then
8          $s'' \leftarrow$  “if  $e_1 == e_2 \wedge \dots \wedge e_1 == e_k$  { $s''$ }”
9          $B' \leftarrow B' - \{\langle x, e_2 \rangle, \dots, \langle x, e_k \rangle\}$ 
10      else if  $x$  does not appear in  $B'$  then
11        Create a fresh variable  $t$ 
12        Find  $v$  in  $s$ , where  $x$  is used as its index
13         $s'' \leftarrow$  “for  $t$  in  $v$  { $s''$ }”
14         $B' \leftarrow B' \cup \{\langle x, t \rangle\}$ 
15      if  $b = \langle x', e' \rangle \wedge x' \neq \perp$  then
16         $s'' \leftarrow s''[x'/e']$ 
17         $B' \leftarrow B' - \{b\}$ 
18      for  $\langle x, e \rangle \in B'$  do
19         $s'' \leftarrow s''[x/e]$ 
20 return  $s'$ 

```

Figure 10. The definition of $\text{Rewrite}()$.

5 Implementation

In our SOLYTHESIS implementation, we use Antlr [1] and the Solidity parser [12] to parse standard specifications and Solidity programs. SOLYTHESIS extends the language described in Section 4 to support all Solidity features including contract function calls.

5.1 Function Calls

Because state variables are often updated sequentially in a transaction and the invariant may be temporarily violated during the middle of the transaction, SOLYTHESIS should only check the constraints at the end of each transaction. For function calls, simply inserting those checks at the end of each function may cause those checks to be triggered multiple times during a transaction and result in false positives. To this end, SOLYTHESIS uses a global state variable (i.e., `memory_loc` in Figure 11) to track whether the current function call is the execution entrance of a transaction. The instrumented constraint checks will only execute if it is so.

SOLYTHESIS uses Surya [13] to build the call graph of the smart contract. With the call graph information, SOLYTHESIS

```

1  uint256 memory_loc;
2  function vote(...) {
3    bool is_entry_function = false;
4    uint[] memory x_arr;
5    if (memory_loc == 0) {
6      is_entry_function = true;
7      assembly {
8        let tmp := mload(0x40)
9        mstore(0x40, add(tmp, 640))
10       sstore(memory_loc_slot, tmp)
11       mstore(add(tmp, 0), 19)
12     }
13   }
14   assembly {
15     x_arr := add(sload(memory_loc_slot), 0)
16   }
17   ...
18   if (is_entry_function) {
19     memory_loc = 0;
20   }
21 }

```

Figure 11. Inline assembly to initialize or load a global array.

can obtain the set of functions that are reachable from an entry function. SOLYTHESIS then prunes away the instrumented checks in functions whose entrance status will never change (e.g., it is always an entrance function). Specially, smart contracts in Ethereum can call functions defined in other smart contracts. To guarantee correctness, SOLYTHESIS overestimates that the inter-contract calls will call back any function defined inside the contract when it builds the call graph.

5.2 Global Memory

SOLYTHESIS uses global memory arrays to store free variable bindings for ForAll constraint rules (lines 10-11, 14-15, and 19-20 in Figure 5). It uses the global memory rather than the intra-procedure volatile memory or the blockchain state because 1) a transaction may contain multiple functions and the instrumented code of these functions all need to access these arrays and 2) accessing global memory is significantly cheaper than accessing the blockchain state.

Since the global memory array is not natively supported by Solidity, SOLYTHESIS uses inline EVM assembly to allocate the in-memory array as well as assigning the start location of the array to an array pointer. Figure 11 presents how SOLYTHESIS implements the instrumented array `x_arr` in Figure 5 as a global array pointer. SOLYTHESIS uses the state variable `memory_loc` to store the starting location of the SOLYTHESIS data section in the global memory. When it is zero (i.e., uninitialized default value), it indicates that the current function call is the execution entrance of a transaction. SOLYTHESIS then allocates an array from the global memory and sets `x_arr` to the start of the array (lines 6-12).

```

1  uint opt_13 = ballots[issueId][msg.sender];
2  uint256 opt_14 = weights[issueId][msg.sender];
3  assert(sum_votes[issueId][opt_13] >= opt_14);
4  x_arr.push(issueId);
5  y_arr.push(opt_13);
6  sum_votes[issueId][opt_13] -= opt_14;
7  opt_13 = option;
8  x_arr.push(issueId);
9  y_arr.push(opt_13);
10 sum_votes[issueId][opt_13] += opt_14;
11 assert(sum_votes[issueId][opt_13] >= opt_14);
12 ...
13 // Write back at the end of the function
14 ballots[issueId][msg.sender] = opt_13;

```

Figure 12. Vote function with state variable caches.

Note that Solidity compiler has the following conventions: 1) it allocates global memory consecutively; 2) it uses the global memory location `0x40` to store the next available global memory location; 3) the first location of an array stores the length of the array. The code at lines 8-11 maintains these conventions. Once `x_arr` is initialized, SOLYTHESIS uses `mload` and `mstore` instructions to load and store data from/to it. The initial size of every allocated array is 19 and SOLYTHESIS will expand the array if necessary.

5.3 State Variable Caches

As an optimization, SOLYTHESIS uses stack or volatile memory in Solidity to cache state variable values. Our second observation in Section 2 indicates that blockchain state load/store operations are very expensive, involving disk I/O, cryptographic computations, and multiple EVM instructions. On the other hand, loading/storing values from/to stack or memory only requires a single instruction.

SOLYTHESIS performs static analysis to determine whether the same blockchain state value is accessed multiple times in a function. For every such value, SOLYTHESIS creates a temporary variable to cache it. If multiple functions can access the state value, SOLYTHESIS will create the temporary variable in the global memory and will use the technique similar to Section 5.2 to enable all functions to access it. Otherwise, SOLYTHESIS creates the temporary variable in the stack. At the end of the function, SOLYTHESIS instruments code to write back cached values to the state variable. This enables the optimized code to only execute one state load operation and one state store operation for such a state value. Note that the optimized code is equivalent to the original code because in Ethereum, all transactions are executed sequentially, i.e., the blockchain state can only be read/written by a single transaction at one time.

Figure 12 presents the code after the state variable cache optimization of lines 14-21 from `vote()` in Figure 5. SOLYTHESIS creates two cache variables `opt_13` and `opt_14` for state variable `ballots[issueId][msg.sender]` and

weights[issueId][msg.sender] (line 1-2). And the operations of those two state variables are replaced by cache variables. Note that not all state variables can be cached. For example, SOLYTHESIS does not cache the state variable sum_votes[issueId][opt_13] in line 6 because opt_13 is updated in line 7 and the state variable in line 10 represents a different state variable. At the end of the function, SOLYTHESIS stores opt_13 back to ballots[issueId][msg.sender]. It does not store opt_14 back because opt_14 is not updated.

6 Evaluation

We next evaluate SOLYTHESIS on three representative standards: ERC20 [5] for the fungible token standard, ERC721 [7] for the non-fungible token standard, and ERC1202 [6] for the voting standard. The goal of this evaluation is to measure the overhead introduced by runtime validation and to understand the effectiveness of the SOLYTHESIS instrumentation optimizations. All experiments were performed on an AWS EC2 m5.xlarge virtual machine, which has 4 cores and 16GB RAM. We downloaded and modified Parity v2.6.0 [10] as the Ethereum client to run our experiments. The source code of SOLYTHESIS and all experiment contracts are available².

6.1 Methodology

Benchmark Contracts: We have collected in total 23 smart contracts in ERC20, ERC721, and ERC1202 standards. We focus on these three standards because 1) they are widely adopted, 2) contracts in these standards contain known vulnerabilities, and/or 3) they have sophisticated invariants over map structures with quantifiers. Our benchmark set includes BecToken (an ERC20 contract) and DozerDoll (an ERC721 contract), two contracts that we successfully collect their history transactions from Ethereum. Our benchmark set also includes the ERC1202 example we described in Section 3. Besides the above three contracts, we crawled the top ten ERC-20 contracts and the top ten ERC-721 contracts ranked by Etherscan [8] and included these contracts into our benchmark set, because they correspond to most widely used contracts that represent digital assets with monetary values.

Note that BECToken implements a customized function, batchTransfer, which has an integer overflow error and violates the total supply invariant specified by ERC-20 standard. This error was exploited in 2018 and the market cap of BECToken was evaporated in days [21]. The ERC1202 example has vulnerabilities that we described in Section 3. In our experiments, SOLYTHESIS successfully nullify errors in both of these two contracts. We validate that the instrumented contracts reject our crafted malicious transactions.

Standard Specifications: We specify invariants for ERC20, ERC721, and ERC1202 respectively using the language described in Section 4. ERC1202 is a smart contract standard for

```
1 sum_balance = Map () Sum balances[a] Over (a) Where true;
2 ForAll () Assert totalSupply == sum_balance;
```

Figure 13. Invariant for ERC20.

```
1 sum_tokenCount = Map () Sum _ownedTokensCount[a] Over
  ↪ (a) Where a != 0;
2 ForAll () Assert sum_tokenCount == _allTokens.length;
3 sum_ownersToken = Map (b) Sum 1 Over (a) Where
  ↪ _tokenOwner[a]==b && _tokenOwner[a] != 0;
4 ForAll (a) Assert _ownedTokensCount[a] ==
  ↪ sum_ownersToken[a];
```

Figure 14. Invariant for ERC721.

implementing a voting contract. The example in the ERC1202 standard draft unfortunately contains logic errors. See Section 3 for details.

ERC20: ERC20 is an important smart contract standard that defines the contract interface and specification for implementing fungible digital assets. Figure 13 shows the invariant for ERC20 standard. balances and totalSupply are two state variables in BECToken that store the balance of each address and the total supply of the token. The invariant specifies that the sum of account balances is equal to total supply.

ERC721: Similar to ERC20, ERC721 is a smart contract standard for implementing non-fungible digital assets. Figure 14 shows our invariant for the ERC721 standard. sum_tokenCount and sum_ownersToken are two intermediate variables created by the invariant to track the state of the contract. sum_tokenCount stores the number of minted tokens, which is the number of tokens whose owner is not 0, and sum_ownersToken stores the number of tokens owned by each address. The invariant specifies that sum_tokenCount equals to the length of _allTokens, and the number of tokens owned by each user equals to the values stored in _ownedTokensCount. _allTokens and _ownedTokensCount are two state variables declared in ERC721 smart contracts.

Benchmark Trace Generation: We crawled the Ethereum blockchain and collected the real transaction history of BecToken and DozerDoll. We chose BecToken because its history contains attacks. We chose DozerDoll because it is an ERC721 token, and it has a long transaction history for our experiments. Note that the collected history transactions depend on the blockchain state (e.g., the token balance of accounts), so we cannot reproduce them directly. To address this issue, we first create a mapping that maps real-world addresses to local addresses that are managed by the Parity client. For each transaction, we replaced the addresses of the transaction sender and receiver, as well as addresses in transaction data.

²<https://github.com/Leeleo3x/solythesis-artifact>

For the remaining contracts, we developed a script to generate random transaction traces that exercise core functionalities of ERC20, ERC721, and ERC1202. For ERC20 and ERC721, the generated trace contains mainly transfer transactions. For ERC1202, the generated trace contains transactions that call `createIssue` and `vote` functions repeatedly. Each `createIssue` transaction is accompanied by five `vote` transactions created by different voters respectively.

Contract Instrumentation: We apply SOLYTHESIS to instrument all of the 23 benchmark contracts. For comparison, we also instrument the contracts naively without delta update/checking techniques as the baseline, i.e., the instrumentation iteratively checks invariants at the end of transactions. For each contract, we manually inspected the two instrumented contracts and validated that they are functionally equivalent.

Experiments with PoW Consensus: We apply SOLYTHESIS to instrument all of the 23 benchmark contracts. We then run Parity 2.6.0 to start a single node Ethereum network to measure the overhead of instrumented contracts. For `BecToken` and `DozerToken`, we use the collected Ethereum history trace. For the ERC1202 example contract and other contracts, we use the generated trace. To run a contract on a transaction trace, we initialize the network with the first 5,052,259 blocks downloaded from Ethereum main net. We then feed the transactions in the trace into the network.

In this experiment, we deploy the same smart contract to the blockchain multiple times and take the average results. To address the randomness of the PoW consensus process, we modified Parity client so that Parity generates new blocks at a fixed speed of 1 block per ten seconds, which is the generation speed upper bound Ethereum ever reaches with its difficulty adjustment mechanism. Note the gas usage for each transaction is set to the gas usage of executing the original smart contract, which allows Parity to pack the same number of transactions for both the instrumented and the original smart contract.

We first set the block gas limit to 8,000,000 and make Parity to pack as many transactions as possible for the native contract run. For the two instrumented runs, we make Parity to pack the exactly same set of transactions per block ignoring the gas limit. This is because we want to measure the runtime overhead and we want to avoid the artificial bottleneck of the gas limit. For each run, we monitor the CPU usage and disk IOs of the Parity client for 400 blocks (~1.4 hours). Because the consensus is the performance bottleneck, there is no significant throughput difference between the original contracts and the instrumented contracts for the native contract run and the SOLYTHESIS instrumentation. We therefore measure the resource consumption as the instrumentation overhead in this set of experiments. For the baseline instrumentation, the runtime overhead grows as the contract state grows with transactions. We also measure how many blocks

after which the baseline instrumentation overhead causes Parity to slow down the block generation.

Experiments without PoW Consensus: We modified Parity to remove the proof of work consensus, but preserved all required computations and storage operations for the transaction execution. To evaluate the potential overhead of SOLYTHESIS when the consensus is no longer the performance bottleneck, we run the SOLYTHESIS instrumented contracts on the modified Parity client to measure the overhead. To evaluate the SOLYTHESIS instrumentation optimizations, we also run naively instrumented contracts for comparison. Specifically, we compare the transaction throughput of the original contracts and the instrumented contracts. Note that in these experiments, we initialize the Parity client with the first 5,052,259 blocks from the Ethereum main net.

6.2 Results with Consensus

Figure 15 shows the resources consumed by Parity for all benchmark contracts. For each experiment, we measure the performance of the SOLYTHESIS instrumented smart contract (**S**), the baseline smart contract (**B**), and the original smart contract (**O**) respectively. Rows 2-4 present the average CPU usage of Parity. Rows 5-7 present the average data written to the disk per second by Parity. For the original version and the SOLYTHESIS instrumented version of each contract, we run the Parity to pack 400 blocks of transactions in our experiments.

Our results show that for all contracts, the transaction execution consumes only a very small portion of the CPU and disk resources. We observe that the CPU usage of Parity is lower than 10% for 95% of the time and the average CPU usage of Parity is lower than 4% for the SOLYTHESIS instrumented and original benchmarks. The overhead introduced by SOLYTHESIS is negligible considering the current capacity of CPU and disk storage devices and the cost of solving proof-of-work. The results validate our previous observation again that the transaction execution is not the bottleneck of the Ethereum blockchain system. Thus, adding runtime validation will not downgrade the performance of Ethereum, but improves the security significantly.

On the other hand, our results show that the CPU and disk overhead introduced by the baseline instrumentation is significant. It increases over time as the contract state grows, because it often iterates over an entire ledger map to check the invariants. In our experiments, we observed that for some contracts the baseline may cause the transaction execution to overtake the consensus to become the new bottleneck, i.e., at a certain point it cannot process the packed transactions in each block even in 10 seconds (i.e., the block generation interval). Row 8 in Figure 15 presents the number of packed blocks for the baseline instrumentation of each contract before the transaction processing time of each block exceeds 10 seconds. For example, Parity cannot process one block

		BEC	USDT	ZRX	THETA	INB	HEDG	DAI	EKT	XIN	HOT	SWP
CPU (%)	S	1.27	3.35	3.49	3.14	3.08	3.40	3.33	3.33	3.43	3.33	3.45
	B	4.15	23.2	19.6	19.5	19.2	18.8	23.3	19.8	18.6	19.2	19.4
	O	1.19	3.40	3.33	2.92	2.90	3.40	3.27	3.32	3.28	3.86	3.27
Disk (KB/s)	S	43.7	96.6	95.5	95.5	95.9	96.2	95.6	95.8	95.3	95.7	96.5
	B	78.0	149.1	148.1	147.4	149.4	148.4	147.7	147.6	148.8	148.5	150.0
	O	44.1	95.9	95.3	95.3	95.7	96.0	95.4	95.3	95.1	95.3	95.9
Blocks	B	995	624	761	813	732	789	783	784	628	784	787

		DOZ	MCHH	CC	CLV	LAND	CARDS	KB	TRINK	PACKS	BKC	EGG	VOTE
CPU (%)	S	1.26	2.39	1.98	2.32	2.16	2.00	2.27	1.99	1.98	1.89	1.96	2.18
	B	2.20	8.01	7.68	7.98	7.96	7.97	8.20	7.55	8.17	7.96	7.53	6.77
	O	1.29	2.01	2.16	2.25	2.09	2.30	2.25	2.14	1.82	1.82	1.69	2.19
Disk (KB/s)	S	57.5	80.00	81.40	81.03	81.60	80.50	81.80	80.66	88.76	82.35	82.87	40.0
	B	67.4	126.5	126.4	124.2	121.3	125.1	123.1	121.0	127.4	124.9	121.4	44.6
	O	53.8	68.73	73.64	68.68	72.01	71.87	70.80	72.56	77.57	71.15	68.27	35.3
Blocks	B	6536	359	359	359	359	359	359	359	359	359	322	240

Figure 15. Resource usage for Parity client. "S" corresponds to SOLYTHESIS results. "O" corresponds to original contract results. "B" corresponds to baseline results.

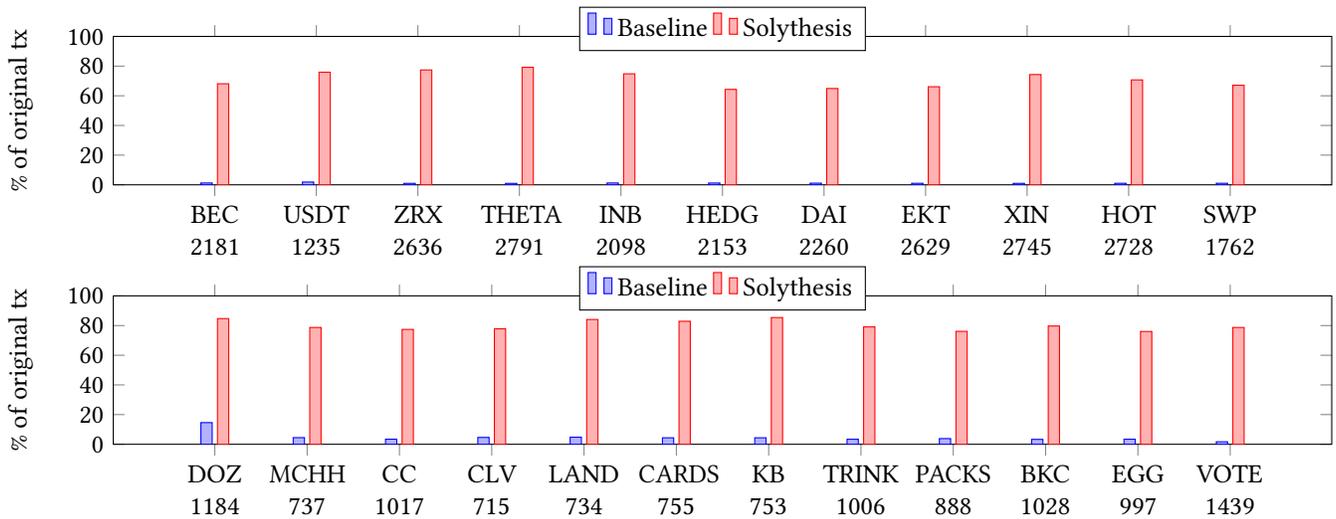


Figure 16. Overhead comparison with respect to the original contract for top ERC20, ERC721 and ERC1202 contracts.

within 10 seconds after 624 blocks for the USDT contract with the baseline instrumentation. Note that we do not observe such significant slowdowns in the SOLYTHESIS instrumented contracts and the native contracts.

6.3 Results without Consensus

To understand the overhead of the SOLYTHESIS instrumentation under fast consensus protocols, we run experiments on Parity when we turn off the consensus layer. We also compare the overhead of SOLYTHESIS with the baseline instrumentation algorithm (which naively performs iterative checks at the end of each transaction call) to evaluate the effectiveness of our optimizations.

Figure 16 presents our experimental results. The X axis corresponds to different smart contracts. The labels in the X axis include both the smart contract name and the TPS of the original contract. The Y axis corresponds to the transaction throughput in the number of transactions processed by Parity per second (TPS). The Y axis is normalized to the TPS of the original smart contracts. Red bars in Figure 16 correspond to the throughput results of SOLYTHESIS, while blue bars correspond to the results of the baseline algorithm.

Our results show that even in the extreme cases where the consensus is no longer the performance bottleneck, the instrumentation overhead of SOLYTHESIS would still be acceptable. The average TPS slowdown caused by the SOLYTHESIS

instrumentation in this set of experiments is 24%. Our results also highlight the importance of the delta update and delta check techniques in SOLYTHESIS. Without those optimizations, the naive instrumentation slows down the transaction throughput by two orders of magnitude. The SOLYTHESIS instrumentation is significantly faster because the delta update and delta check techniques enable the runtime check to only access a small fixed number of blockchain state entries per transaction. As the contract state grows, the instrumentation overhead of the baseline version grows linearly, while the overhead of SOLYTHESIS remains stable.

7 Discussion and Future Directions

Our experimental results demonstrate that runtime validation is much more affordable in smart contracts than in many other domains. Because of the performance bottlenecks at the consensus layer and the storage layer, lightweight runtime instrumentations can have small or even negligible overhead. Our results reveal several future research directions on how to improve the security of smart contracts and blockchain platforms.

New Languages with Runtime Validation: To secure smart contracts, people are designing new smart contract languages that can eliminate certain classes of errors at the compile time, at the cost of limiting language expressiveness [22, 26, 57]. Our results imply that making the difficult trade-off between the correctness and the expressiveness may not be necessary. One possible future direction is to design new languages that can better utilize rigorous runtime validation to enforce correctness and security instead.

Static Analysis and Verification: Developing static analysis and verification techniques to secure smart contracts is both an active research topic and an industrial trend [3, 11, 20, 32, 36, 39, 42, 45, 46, 48, 52, 61, 63]. Like similar techniques in traditional programs, static analysis techniques often have inaccurate results, while verification techniques typically require human intervention. One possible future direction is to combine runtime validation with these static analysis and verification techniques. Because of the inexpensive cost, runtime validation can act as a backup technique to cover scenarios that static analysis techniques fail to fully analyze or that verification techniques cannot fully prove.

Runtime Validation in Blockchain VM: SOLYTHESIS implements runtime checks via Solidity source code instrumentation. It is possible to implement some of the runtime checks in the blockchain virtual machine to further reduce the overhead, but this would require a hard fork for existing blockchains like Ethereum to adopt the technique.

New Gas Mechanism: Each EVM instruction charges gas and the fee of an Ethereum smart contract transaction is determined by the total gas the transaction consumes. In our experiments, the average gas overhead of the instrumented contracts is 77.8%, which is significantly higher than

the actual resource consumption overhead (which is negligible). One possible explanation is that the gas schedule in Ethereum does not correctly reflect the execution cost of each EVM instruction. The gas overhead would cause the users of the instrumented contracts to pay additional transaction fees. In the light of our results, we believe Ethereum and future blockchain systems should adopt a more flexible gas mechanism to facilitate runtime validation techniques.

8 Related Work

Runtime Validation for Smart Contracts: ContractLarva [28] allows users to supply pre- and post-conditions for smart contract transactions and enforces them at runtime. Unlike SOLYTHESIS, it does not support quantifiers natively. Note that quantifiers are important because many contracts store critical data into ledgers that map each Ethereum address to balance-like values. Without quantifiers, it is not possible to write many meaningful global invariants (e.g., all evaluated invariants in our experiments) for such ledger data structures. Users may write looped pre/post-conditions instead to simulate such invariants in ContractLarva, but that would be equivalent to the naive approach in our paper and would cause very high overhead. The ContractLarva paper evaluates the system on one sample ERC-20 contract with no end-to-end runtime overhead result.

Sereum [55] is a new design and implementation of Ethereum virtual machine which tracks the storage operations of each transaction to prevent re-entrancy attacks. Modifying the EVM implementation to perform runtime checks may further reduce the overhead, but it would require a blockchain hard-fork. On the other hand, SOLYTHESIS inserts runtime checks at the Solidity level, which does not require any modification of the existing blockchain system.

SODA [24] is an EVM online detection platform in which users can develop monitor programs for deployed smart contracts. It acts as a plugin to the full node of Ethereum to fetch the states of smart contracts. The monitor programs then run on the fetched states to detect errors and anomalies. Unlike SOLYTHESIS which turns the offending transactions into no-ops, SODA cannot mitigate the detected errors, i.e., the offending transactions will be processed by Ethereum and the attacks will still succeed.

Static Analysis and Verification for Smart Contracts: There is a rich body of work on detecting vulnerable smart contracts with different techniques such as symbolic execution [36, 39, 42, 45, 48, 51–53, 63], fuzzing [31, 34], domain-specific static analysis [61], and formal verification [20, 25, 32, 46]. Oyente [45] detects transaction-order dependency attacks, re-entrance attacks, and mishandled exception attacks using symbolic execution. Verx [53] uses delayed abstraction to detect and verify temporal safety properties automatically. He et al. present a new fuzzing technique that learns from symbolic execution traces to achieve

both high coverage and high speed [31]. Securify presents a domain-specific formal verification technique that translates Solidity smart contract into Datalog and verifies security properties such as restricted storage writes and ether transfers [61].

SOLYTHESIS differs from these previous static analysis, fuzzing, and symbolic execution approaches in that it inserts runtime checks to enforce user-specified invariants. Unlike these approaches, SOLYTHESIS does not suffer from false positives and false negatives. Compared to most formal verification approaches, SOLYTHESIS is fully automated and does not require human intervention. Securify is an automated verification tool using SMT solvers, but it may not scale to complicated contracts due to the potential SMT expression explosion problem. Also, unlike SOLYTHESIS, it cannot support sophisticated constraints like quantifiers.

Runtime Validation in Other Domains: Runtime validation techniques have been used to enforce security constraints in other general-purpose languages [15, 17, 18, 35, 58]. Tracematches [15] and JavaMop [35] focus on the expressiveness of the specification language. QEA [17] and DITTO [58] allow users to write potentially complicated specifications while allowing for optimizations.

Runtime checks are also useful for detecting and mitigating memory errors in C/C++ applications [14, 19, 23, 33, 40, 49, 59]. Control-flow integrity is a runtime mechanism to prevent attackers from hijacking the control flow of a program [14, 29]. Simpson and Barua enforce both spatial and temporal memory safety of C programs without introducing high overhead by modelling temporal errors as spatial errors and removing redundant checks [59]. Similarly, Frama-C generates runtime memory monitors automatically to check E-ACSL specifications [40]. Chandola et al. describe that patterns in data that do not conform to expected behavior can help the system detect intrusions, frauds and faults [23]. Berger and Zorn, for example, implement a runtime system that executes multiple replicas of the same application simultaneously to detect memory errors [19]. RCV uses lightweight binary instrumentations to enable applications to safely recover from null-dereference and divide-by-zero errors with negligible overhead [44]. Researchers have also used runtime checks to detect side-channel attacks [64] and isolate errors [62].

Comparing to other general-purpose languages, the uniqueness of the smart contract setting is that blockchain state access operations are significantly more expensive than other types of operations, and they dominate the EVM execution as shown in Section 2. This phenomenon drives the design of Solythesis. The delta update techniques in Solythesis focuses on minimizing the usage of state access operations as much as possible, while the runtime validation techniques in other tools have a different design focus.

Blockchain Consensus: The consensus layer is the primary performance bottleneck for many existing blockchain

systems including Ethereum. Developing fast consensus protocols for blockchain is an active research topic. Several new high-performance consensus protocols have been proposed and implemented [16, 30, 38, 43, 47, 54, 66]. Many of these new protocols can support hundreds to thousands of transactions per second. Our experimental results show that SOLYTHESIS will have low overhead when applying on future blockchain systems built on these fast consensus protocols.

New Language Design: Many new programming languages have been recently proposed to improve smart contract security. Scilla [57] is a low-level smart contract language with a refined type system that is easy to verify. Move [22] introduces resource types and uses linear logic to enforced access control policies for digital assets. Obsidian [26] uses tpestate and linear type to enforce static checks. The trend of these new languages is to sacrifice expressiveness (e.g., no longer Turing-complete) to gain correctness or security guarantees. Interestingly, our results reveal an alternative path — one possible future direction is to design new languages that can better utilize rigorous runtime validation to enforce correctness and security.

K-framework is a rewrite-based semantic framework that allows developers to specify semantics of a programming language formally [56], which could be used to design and implement such new languages that facilitate runtime validation. KEVM [32] defines the semantic of EVM in \mathbb{K} and verifies the smart contract against user-defined specifications. IELE [37] presents a smart contract virtual machine with a formal specification described in \mathbb{K} which achieves similar performance as the EVM and provides verifiability.

9 Conclusion

Runtime validation is an effective and efficient approach to secure smart contracts. With the delta update and delta check techniques, SOLYTHESIS can efficiently enforce powerful invariants with quantifiers for the state of a smart contract. The experimental results of SOLYTHESIS validate our hypothesis that because the transaction execution is not the performance bottleneck in Ethereum, the overhead of runtime validation, which is often too expensive for other domains, is in fact negligible for smart contracts. In the light of our results, we believe that new languages, new analyses, and new virtual machine designs that can further exploit rigorous runtime validation are promising future directions to secure smart contracts.

Acknowledgments

We thank Ming Wu and the anonymous reviewers for their insightful comments on the early draft of this paper. This research was supported by Connaught Fund #507141 and Tri-Council Bridge Funding from University of Toronto. Note that an early version of this paper appears in arXiv [41].

References

- [1] [n. d.]. ANTLR: ANOther Tool for Language Recognition. <https://www.antlr.org/>.
- [2] [n. d.]. Beauty Chain: The world's first blockchain platform dedicated to the beauty ecosystem. <http://www.beauty.io/>.
- [3] [n. d.]. Certik: World's most advanced formal verification technology for smart contracts. <https://www.certik.org/>.
- [4] [n. d.]. CryptoDozer. <https://cryptodozer.io/>.
- [5] [n. d.]. EIP 1202: Token Standard. <https://eips.ethereum.org/EIPS/eip-20>.
- [6] [n. d.]. EIP 1202: Voting Standard. <https://eips.ethereum.org/EIPS/eip-1202>.
- [7] [n. d.]. EIP 721: Non-Fungible Token Standard. <https://eips.ethereum.org/EIPS/eip-721>.
- [8] [n. d.]. Ethereum (ETH) Blockchain Explorer. <https://etherscan.io/>.
- [9] [n. d.]. Ethereum Gas Limit History. <https://etherscan.io/chart/gaslimit>.
- [10] [n. d.]. Parity: The fastest and most advanced Ethereum client. <https://www.parity.io/ethereum/>.
- [11] [n. d.]. Quantstamp: Leaders in blockchain security and solutions. <https://quantstamp.com/>.
- [12] [n. d.]. solidity-parser-antlr. <https://github.com/federicobond/solidity-parser-antlr>.
- [13] [n. d.]. Surya, The Sun God: A Solidity Inspector. <https://github.com/ConsenSys/surya>.
- [14] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353. <https://doi.org/10.1145/1102120.1102165>
- [15] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2005. Adding Trace Matching with Free Variables to AspectJ. *SIGPLAN Not.* 40, 10 (Oct. 2005), 345–364. <https://doi.org/10.1145/1103845.1094839>
- [16] Vivek Kumar Bagaria, Sreeram Kannan, David Tse, Giulia C. Fanti, and Pramod Viswanath. 2018. Deconstructing the Blockchain to Approach Physical Limits. *CoRR* abs/1810.08092 (2018). arXiv:1810.08092 <http://arxiv.org/abs/1810.08092>
- [17] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. 2012. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM 2012: Formal Methods, Dimitra Giannakopoulou and Dominique Méry* (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–84.
- [18] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. 2011. MONPOLY: Monitoring Usage-Control Policies. In *RV*.
- [19] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 158–168. <https://doi.org/10.1145/1133981.1134000>
- [20] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguélin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96. <https://doi.org/10.1145/2993600.2993611>
- [21] John Biggs. 2018. Overflow error shuts down token trading. <https://techcrunch.com/2018/04/25/overflow-error-shuts-down-token-trading/>.
- [22] Sam Blackshear, Evan Chengand David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2019. *Move: A Language With Programmable Resources*. Technical Report. <https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf>
- [23] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15, 58 pages. <https://doi.org/10.1145/1541880.1541882>
- [24] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. <https://doi.org/10.14722/ndss.2020.24449>
- [25] Xiaohong Chen, Daejun Park, and Grigore Roşu. 2018. A Language-Independent Approach to Smart Contract Verification. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 405–413.
- [26] Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ArXiv* abs/1909.03523 (2019).
- [27] Phil Daian. 2016. Analysis of the DAO exploit. <http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>.
- [28] J. Ellul and G. J. Pace. 2018. Runtime Verification of Ethereum Smart Contracts. In *2018 14th European Dependable Computing Conference (EDCC)*. 158–163. <https://doi.org/10.1109/EDCC.2018.00036>
- [29] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 901–913. <https://doi.org/10.1145/2810103.2813646>
- [30] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 51–68. <https://doi.org/10.1145/3132747.3132757>
- [31] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. 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 '19)*. ACM, New York, NY, USA, 531–548. <https://doi.org/10.1145/3319535.3363230>
- [32] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 204–217. <https://doi.org/10.1109/CSF.2018.00022>
- [33] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2373–2387. <https://doi.org/10.1145/3133956.3134062>
- [34] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 259–269. <https://doi.org/10.1145/3238147.3238177>
- [35] Dongyun Jin, Patrick Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. *Proceedings - International Conference on Software Engineering* (06 2012), 1427–1430. <https://doi.org/10.1109/ICSE.2012.6227231>
- [36] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. NDSS.
- [37] Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Florin Şerbănuţă, Yi Zhang, Daniele Filaretti, Virgil Şerbănuţă, Ralph Johnson,

- and Grigore Roşu. 2019. IELE: A Rigorously Designed Language and Tool Ecosystem for the Blockchain. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 593–610.
- [38] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. 583–598. <https://doi.org/10.1109/SP.2018.000-5>
- [39] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2018. Exploiting The Laws of Order in Smart Contracts. *CoRR* abs/1810.11605 (2018). arXiv:1810.11605 <http://arxiv.org/abs/1810.11605>
- [40] Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. 2013. An Optimized Memory Monitoring for Runtime Assertion Checking of C Programs. In *Runtime Verification*, Axel Legay and Saddek Bensalem (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–182.
- [41] Ao Li, Jemin Andrew Choi, and Fan Long. 2019. Securing Smart Contract On The Fly. arXiv:cs.PL/1911.12555
- [42] Ao Li and Fan Long. 2018. Detecting Standard Violation Errors in Smart Contracts. *CoRR* abs/1812.07702 (2018). arXiv:1812.07702 <http://arxiv.org/abs/1812.07702>
- [43] Chenxing Li, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2018. Scaling Nakamoto Consensus to Thousands of Transactions per Second. *CoRR* abs/1805.03870 (2018). arXiv:1805.03870 <http://arxiv.org/abs/1805.03870>
- [44] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 227–238. <https://doi.org/10.1145/2594291.2594337>
- [45] 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 (CCS '16)*. ACM, New York, NY, USA, 254–269. <https://doi.org/10.1145/2976749.2978309>
- [46] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiri, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. *CoRR* abs/1901.01292 (2019). arXiv:1901.01292 <http://arxiv.org/abs/1901.01292>
- [47] Libra Association Members. 2019. *An Introduction to Libra*. Technical Report. https://libra.org/en-US/wp-content/uploads/sites/23/2019/06/LibraWhitePaper_en_US.pdf
- [48] Bernhard Mueller. [n. d.]. Mythril Classic: Security analysis tool for Ethereum smart contracts. <https://github.com/ConsenSys/mythril-classic>.
- [49] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 190–208. <https://doi.org/10.4230/LIPIcs.SNAPL.2015.190>
- [50] Satoshi Nakamoto. [n. d.]. Bitcoin: A peer-to-peer electronic cash system," <http://bitcoin.org/bitcoin.pdf>.
- [51] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018). arXiv:1802.06038 <http://arxiv.org/abs/1802.06038>
- [52] Trail of Bits. [n. d.]. Manticore: Symbolic Execution for Humans. <https://github.com/trailofbits/manticore>.
- [53] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachler-Cohen, and Martin Vechev. [n. d.]. VerX: Safety Verification of Smart Contracts. ([n. d.]).
- [54] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gun Sirer. 2019. Scalable and Probabilistic Leaderless BFT Consensus through Metastability. *CoRR* abs/1906.08936 (2019). arXiv:1906.08936 <http://arxiv.org/abs/1906.08936>
- [55] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2018. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. arXiv:cs.CR/1812.05934
- [56] G. Rosu. 2017. *K: A semantic framework for programming languages and formal analysis tools*. 186–206. <https://doi.org/10.3233/978-1-61499-810-5-186>
- [57] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 185 (Oct. 2019), 30 pages. <https://doi.org/10.1145/3360611>
- [58] Ajeet Shankar and Rastislav Bodik. 2007. DITTO: Automatic Incrementalization of Data Structure Invariant Checks (in Java). *SIGPLAN Not.* 42, 6 (June 2007), 310–319. <https://doi.org/10.1145/1273442.1250770>
- [59] M. S. Simpson and R. K. Barua. 2010. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 199–208. <https://doi.org/10.1109/SCAM.2010.15>
- [60] Yonatan Sompolsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *Financial Cryptography and Data Security*, Rainer Böhme and Tatsuaiki Okamoto (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 507–527.
- [61] Petar Tsankov, Andrei Dan, Dana Drachler-Cohen, Arthur Gervais, Florian Bunzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 67–82. <https://doi.org/10.1145/3243734.3243780>
- [62] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 203–216. <https://doi.org/10.1145/173668.168635>
- [63] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 189 (Oct. 2019), 29 pages. <https://doi.org/10.1145/3360615>
- [64] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-wasm: Type-driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.* 3, POPL, Article 77 (Jan. 2019), 29 pages. <https://doi.org/10.1145/3290390>
- [65] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.
- [66] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. 2018. OHIE: Blockchain Scaling Made Simple. *CoRR* abs/1811.12628 (2018). arXiv:1811.12628 <http://arxiv.org/abs/1811.12628>