VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts

Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, Hakjoo Oh*
Department of Computer Science and Engineering
Korea University

***Abstract*—We present VERISMART, a highly precise verifier for ensuring arithmetic safety of Ethereum smart contracts. Writing safe smart contracts without unintended behavior is critically important because smart contracts are immutable and even a single flaw can cause huge financial damage. In particular, ensuring that arithmetic operations are safe is one of the most important and common security concerns of Ethereum smart contracts nowadays. In response, several safety analyzers have been proposed over the past few years, but state-of-the-art is still unsatisfactory; no existing tools achieve high precision and recall at the same time, inherently limited to producing annoying false alarms or missing critical bugs. By contrast, VERISMART aims for an uncompromising analyzer that performs exhaustive verification without compromising precision or scalability, thereby greatly reducing the burden of manually checking undiscovered or incorrectly-reported issues. To achieve this goal, we present a new domain-specific algorithm for verifying smart contracts, which is able to automatically discover and leverage transaction invariants that are essential for precisely analyzing smart contracts. Evaluation with real-world smart contracts shows that VERISMART can detect all arithmetic bugs with a negligible number of false alarms, far outperforming existing analyzers.**

I. INTRODUCTION

Safe smart contracts are indispensable for trustworthy blockchain ecosystems. Blockchain is widely recognized as one of the most disruptive technologies and smart contracts lie at the heart of this revolution (e.g., [1], [2]). Smart contracts are computer programs that run on blockchains in order to automatically fulfill agreed obligations between untrusted parties without intermediaries. Unfortunately, despite their potential, smart contracts are more likely to be vulnerable than traditional programs because of their unique characteristics such as openness and immutability [3]. As a result, unsafe smart contracts are prevalent and are increasingly becoming a serious threat to the success of the blockchain technology. For example, recent infamous attacks on the Ethereum blockchain such as the DAO [4] and the Parity Wallet [5] attacks were caused by unsafe smart contracts.

In this paper, we present VERISMART, a fully automated safety analyzer for verifying Ethereum smart contracts with a particular focus on arithmetic safety. We focus on detecting arithmetic bugs such as integer over/underflows and division-by-zeros because smart contracts typically involve lots of arithmetic operations and they are major sources of security vulnerabilities nowadays. For example, arithmetic over/underflows account for 95.7% (487/509) of CVEs assigned to Ethereum smart contracts, as shown in Table I. Even worse, arithmetic bugs, once exploited, are likely to cause significant but unexpected financial damage (e.g., the integer overflow in the SmartMesh contract [6] explained in Section II). Our goal is to detect all arithmetic bugs before deploying smart contracts on the blockchain.

TABLE I
STATISTICS ON CVE-REPORTED SECURITY VULNERABILITIES OF ETHEREUM SMART CONTRACTS (AS OF MAY. 31, 2019)

Arithmetic Over/underflow	Bad Randomness	Access Control	Unsafe Input Dependency	Others	Total
487 (95.7 %)	10 (1.9 %)	4 (0.8 %)	4 (0.8 %)	4 (0.8%)	509

Unlike existing techniques, VERISMART aims to be a truly practical tool by performing automatic, scalable, exhaustive, yet highly precise verification of smart contracts. Recent years have seen an increased interest in automated tools for analyzing arithmetic safety of smart contracts [7], [8], [9], [10], [11], [12]. However, existing tools are still unsatisfactory. A major weakness of bug-finding approaches (e.g., [7], [9], [8], [10]) is that they are likely to miss fatal bugs (i.e., resulting in false negatives), because they do not consider all the possible behaviors of the program. On the other hand, verification approaches (e.g., [11], [12]) are exhaustive and therefore miss no vulnerabilities, but they typically do so at the expense of precision (i.e., resulting in false positives). In practice, both false negatives and positives burden developers with error-prone and time-consuming process for manually verifying a number of undiscovered issues or incorrectly reported alarms. VERISMART aims to overcome these shortcomings of existing approaches by being exhaustive yet precise.

To achieve this goal, we present a new verification algorithm for smart contracts. The key feature of the algorithm, which departs significantly from the existing analyzers for smart contracts [7], [8], [9], [10], [11], [12], is to automatically discover domain-specific invariants of smart contracts during the verification process. In particular, our algorithm automates the discovery of *transaction invariants*, which are distinctive properties of smart contracts that hold under arbitrary interleaving of transactions and enable to analyze smart contracts exhaustively without exploring all program paths separately. A technical challenge is to efficiently discover precise invariants

*Corresponding author: Hakjoo Oh, hakjoo_oh@korea.ac.kr

from the huge search space. We propose an effective algorithm tailored for typical smart contracts, which iteratively generates and validates candidate invariants in a feedback loop akin to the CEGIS (counter example-guided inductive synthesis) framework [13], [14], [15]. Our algorithm is general and can be used for analyzing a wide range of safety properties of smart contracts besides arithmetic safety.

Experimental results show that our algorithm is much more effective than existing techniques for analyzing Ethereum smart contracts. We first evaluated the effectiveness of VERISMART by comparing it with four state-of-the-art bug-finders: OSIRIS [7], OYENTE [9], MYTHRIL [8], and MANTICORE [10]. An in-depth study on 60 contracts that have CVE vulnerabilities shows that VERISMART detects all known vulnerabilities with a negligible false positive rate (0.41%). By contrast, existing bug-finders failed to detect a large amount ($> 29.3\%$) of known vulnerabilities with higher false positive rates ($> 5.4\%$). We also compared VERISMART with two state-of-the-art verifiers, ZEUS [11] and SMTCHECKER [12]. The results show that VERISMART is significantly more precise than them thanks to its ability to discover transaction invariants of smart contracts automatically.

Contributions: Our contributions are as follows:

- We present a new verification algorithm for smart contracts (Section III). This is the first CEGIS-style algorithm that leverages transaction invariants automatically during the verification process.
- We provide VERISMART, a practical implementation of our algorithm that supports the full Solidity language, the de facto standard programming language for writing Ethereum smart contracts.
- We provide in-depth evaluation of VERISMART in comparison with six analyzers [7], [9], [8], [10], [11], [12]. All experimental results are reproducible as we make our tool and data publicly available.[1]

II. MOTIVATING EXAMPLES

In this section, we illustrate central features of VERISMART with examples. We use three real-world smart contracts to highlight key aspects of VERISMART that differ from existing analyzers.

Example 1: Figure 1 shows a simplified function from the SmartMesh token contract (CVE-2018-10376). In April 2018, an attacker exploited a vulnerability in the function and succeeded to create an extremely large amount of unauthorized tokens ($\approx 5{\cdot}10^{57}$ USD). This vulnerability, named proxyOverflow, was due to unexpected integer overflow.

The `transferProxy` function is responsible for transferring a designated amount of tokens (`value`) from a source address (`from`) to a destination address (`to`) while paying transaction fees (`fee`) to the message sender (`msg.sender`). The core functionality is implemented at lines 8–10, where the recipients' balances (`balance[to]` and `balance[msg.sender]`) are increased (lines 8 and 9) and the sender's balance (`balance[from]`) is decreased by the same amount of the sent tokens at line 10.

```
1  function transferProxy (address from, address to, uint
       value, uint fee) {
2    if (balance[from] < fee + value) revert();
3
4    if (balance[to] + value < balance[to] ||
5      balance[msg.sender] + fee < balance[msg.sender])
6        revert();
7
8    balance[to] += value;
9    balance[msg.sender] += fee;
10   balance[from] -= value + fee;
11 }
```

Fig. 1. A vulnerable function from SmartMesh (CVE-2018-10376).

Note that the developer is aware of the risks of integer over/underflows and has made effort to avoid them. The conditional statement at line 2 checks whether the sender's balance (`balance[from]`) is greater than or equal to the tokens to be sent (`fee+value`), aiming to prevent integer underflow at line 10. The guard statements at lines 4 and 5 check that the recipients' balances are valid after the transaction, intending to prevent integer overflows at lines 8 and 9, respectively.

However, the contract still has a loophole at line 2. The expression `fee+value` inside the conditional statement may cause integer overflow, which enables the token sender to send more money than (s)he has. Suppose all accounts initially have no balances, i.e., `balance[from]=0`, `balance[to]=0`, and `balance[msg.sender]=0`, and the function is invoked with the arguments `value=0x8ff...ff` and `fee=0x700...01`, where 256-bit unsigned integer variables (`value` and `fee`) are represented in hexadecimal numbers comprised of 64 digits (e.g., `value` has 63 `f`s and one `8`). Suppose further the two unspecified address values are given as the same but different from the sender's (i.e., `from` $=$ `to` $\neq$ `msg.sender`). These crafted inputs then make the sanity checks at lines 2–6 powerless (i.e., the three conditions at lines 2, 4, and 5 are all false because `fee+value` $=$ `0x8ff...ff` $+$ `0x700...01` $=$ `0` and `balance[to]` $=$ `balance[msg.sender]` $=$ `0`). Therefore, lines 8–10 for token transfer are executed unexpectedly, creating a huge amount of tokens from nothing (i.e., `balance[to]` $=$ `balance[from]` $=$ `0x8ff...ff` and `balance[msg.sender]` $=$ `0x700...01`.

This accident could have been prevented by VERISMART, as it pinpoints the vulnerability at line 2. Indeed, VERISMART is an exhaustive verifier, aiming to detect all arithmetic issues in smart contracts. By contrast, inexhaustive bug-finders are likely to miss critical vulnerabilities. For example, among the existing bug-finders [7], [9], [8], [10], only OSIRIS [7] is able to find the vulnerability. MYTHRIL [8] and OYENTE [9] fail to detect the well-known proxyOverflow vulnerability.

Example 2: Figure 2 shows the `multipleTransfer` function adapted from the Neo Genesis Token contract (CVE-2018-14006). The function has a similar vulnerability to that of the first example. At line 3 in Figure 2, it prevents the underflow possibility of the token sender's account but does

[1] http://prl.korea.ac.kr/verismart

```
1  function multipleTransfer(address[] to, uint value) {
2   require(value * to.length > 0);
3   require(balances[msg.sender] >= value * to.length);
4   balances[msg.sender] -= value * to.length;
5   for (uint i = 0; i < to.length; ++i) {
6     balances[to[i]] += value;
7   }
8  }
```

Fig. 2. A vulnerable function from Neo Genesis Token (CVE-2018-14006).

not protect the overflow of the tokens to be sent (`value * to.length`), which is analogous to the situation at line 2 of Figure 1. That is, in a similar way, an attacker can send huge amounts of tokens to any users by spending only few tokens [16].

Despite the similarity between vulnerabilities in Example 1 and 2, bug-finders have no guarantees of consistently finding them. For example, OSIRIS, which succeeded to detect the vulnerability in Example 1, now fails to report the similar bug in Example 2. The other bug-finders are ineffective too; MYTHRIL does not report any issues and OYENTE obscurely reports that the entire function body is vulnerable without specifying certain operations. On the other hand, VERISMART reliably reports that the expression `value * to.length` at lines 2–4 would overflow.

One of the main reasons for the unstable results of bug-finders is that they rely heavily on a range of heuristics to avoid false positives (e.g., see [7]). Though heuristics are good at reducing false positives, the resulting analyzer is often very brittle; even small changes in programs may end up with missing fatal vulnerabilities as shown in Example 1 and 2, which is particularly undesirable for safety-critical software like smart contracts.

Example 3: Figure 3 shows a simplified version of the contract, called BTX. The program has two global state variables: `balance` stores balances of each account address (line 2), and `totalSupply` is the total amount of the supplied tokens (line 3). The constructor function initializes `totalSupply` with 10000 tokens (line 6), and gives the same amount of tokens to the creator of the contract (line 7). The `transfer` function sends `value` tokens from the transaction message sender's account to the recipient's account (lines 12–13), if it does not incur the underflow in the message sender's balance (line 11). The `transferFrom` function is similar to `transfer` with an exception to the order of performing addition and subtraction.

The contract has four arithmetic operations at lines 12, 13, 18, and 19, all of which are free of integer over/underflows. However, it is nontrivial to see why they are all safe. In particular, the safety of the two addition operations at lines 13 and 18 is tricky, because there are no direct safety-checking statements in each function. To see why they do not overflow, we need to discover the following two *transaction invariants* that always hold no matter how the transactions (`transfer` and `transferFrom`) are interleaved:

```
1  contract BTX {
2    mapping (address => uint) public balance;
3    uint public totalSupply;
4
5    constructor () {
6      totalSupply = 10000;
7      balance[msg.sender] = 10000;
8    }
9
10   function transfer (address to, uint value) {
11     require (balance[msg.sender] >= value);
12     balance[msg.sender] -= value;
13     balance[to] += value; // Safe
14   }
15
16   function transferFrom (address from, address to, uint
         value) {
17     require (balance[from] >= value);
18     balance[to] += value; // Safe
19     balance[from] -= value;
20   }
21 }
```

Fig. 3. Example contract simplified from CVE-2018-13326.

- the sum of all account values is 10000, i.e.,

$$\sum_i \texttt{balance}[i] = 10000, \tag{1}$$

- and computing $\sum_i \texttt{balance}[i]$ does not cause overflow.

By combining these two conditions and the preconditions expressed in the `require` statements at lines 11 and 17, we can conclude that, at lines 13 and 18, the maximum values of both `balance[to]` and `value` are 10000, and thus the expression `balance[to]+value` does not overflow in 256-bit unsigned integer operations.

Since reasoning about the safety in this case is tricky, it is likely for human auditors to make a wrong conclusion that the contract is unsafe. This is in fact what happened in the recent CVE report (CVE-2018-13326)[2]; the CVE report incorrectly states that the two addition operations at lines 13 and 18 are vulnerable and thus the operations may overflow. Unfortunately, existing safety analyzers do not help here. In particular, verifiers, ZEUS [11] and SMTCHECKER [12], are not precise enough to keep track of the implicit invariants such as (1) and therefore cannot prove the safety at lines 13 and 18. Bug-finders OSIRIS and OYENTE also produce false alarms. MYTHRIL does not report any issues, but this does not mean that it proved the absence of vulnerabilities.

By contrast, VERISMART is able to prove that the contract is safe without any false alarms. Notably, VERISMART does so by automatically inferring hidden invariants described above. To our knowledge, VERISMART is the first of its kind, which discovers global invariants of smart contracts and leverages them during the verification process in a fully automated way.

III. VERISMART ALGORITHM

This section describes the verification algorithm of VERISMART. We formally present the algorithm in a general setting, so it can be used for analyzing other safety properties as well beyond our application to arithmetic safety.

[2]https://nvd.nist.gov/vuln/detail/CVE-2018-13326

Language: For brevity, we focus on a core subset of Solidity [17]. However, VERISMART supports the full Solidity language as the extension is discussed in Section IV. Consider the following subset of Solidity:

$$
\begin{array}{lll}
c \in C & ::= & G^*\ F^*, \qquad f \in F \ ::= \ x(y)\{S\} \\
a \in A & ::= & x := E \mid x[y] := E \mid \mathit{assume}(B) \mid \mathit{assert}(B) \\
s \in S & ::= & A \mid \mathit{if}\ B\ S_1\ S_2 \mid \mathit{while}^l\ E\ S \mid S_1; S_2
\end{array}
$$

We assume a single contract c is given, which consists of a sequence of global state variable declarations (G^*) and a sequence of function definitions (F^*), where G and F denote the sets of global variables and functions in the contract, respectively. We assume a constructor function $f_0 \in F$ exists in c. Each function f is defined by a function name (x), argument (y), and a body statement (S). A statement S is an atomic statement (A), a conditional statement, or a while loop. An atomic statement $a \in A$ is an assignment to a variable ($x := E$), an assignment to an array element ($x[y] := E$), an *assume* statement, or an *assert* statement. In our language, we model mapping variables in Solidity as arrays. In our language, *assume* differs from *assert*; while the former models the `require` statements in Solidity and stops execution if the condition evaluates to false, the latter does not affect program semantics. E and B stand for conventional arithmetic and boolean expressions, respectively, where we assume arithmetic expressions produce 256-bit unsigned integers. In our language, loops are annotated with labels (l), and the entry and the exit of each function f are annotated with special labels entry_f and exit_f, respectively. Let *Label* be the set of all labels in the program. We assume each function f has `public` (or `external`) visibility, meaning that all functions in the contract can be called from the outside.

Goal: Our goal is to develop an algorithm that proves or disproves every assertion (which we also call *query*) in the contract. We assume that safety properties to verify are expressed as the *assert* statements in the program. In our application to arithmetic safety, assertions can be automatically generated; for example, for each addition `a+b` and multiplication `a*b`, we generate `assert(a+b>=a)` and `assert(a==0||(a!=0 && (a*b)/a==b))`, respectively.

Notation: We use the lambda notation for functions. For example, $\lambda x.x + 1$ is the function that takes x and returns $x+1$. We write FOL for the set of first-order formulas in the combined theory of fixed-sized bitvectors, arrays with extensionality, and equality with uninterpreted functions. When e is an expression or a formula, we write $e[y/x]$ for the new expression where x gets replaced by y. We write $\mathsf{FV}(e)$ for the set of free variables in e.

A. Algorithm Overview

VERISMART departs significantly from existing analyzers for smart contracts [7], [8], [9], [10], [11], [12], [18], [19], [20], [21] in that VERISMART applies a CEGIS-style verification algorithm that iteratively searches for hidden invariants that are required for verifying safety properties.

```
1  contract RunningExample {
2    uint public n;
3    constructor () { n = 1;}
4    function f () public {
5      assert (n + 1 >= n);
6      n = n + 1;
7      if (n >= 100) { n = 1; }
8    }
9  }
```

Fig. 4. Example contract.

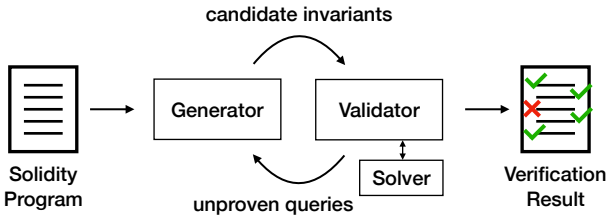

Fig. 5. Algorithm overview.

Invariants of Smart Contracts: We consider two kinds of invariants for smart contracts: transaction and loop invariants. We say a formula is a transaction invariant if it is valid at the end of the constructor and the validity is preserved by the execution of public functions that can be invoked by transactions. Loop invariants are more standard; a formula is an invariant of a loop if the formula is valid at the entry of the loop and is preserved by the loop body. Transaction invariant is global and thus it is a single formula, whereas loop invariants are local and must be separately given for each loop in the program. Thus, our algorithm aims to discover a pair (ψ, μ), where $\psi \in \mathsf{FOL}$ is a transaction invariant and $\mu \in \mathit{Label} \to \mathsf{FOL}$ is a mapping from loop labels to formulas. We write $\bigwedge$ for pointwise conjoining operation between two mappings μ_1 and μ_2, i.e., $\mu_1 \bigwedge \mu_2 = \lambda l \in \mathit{Label}.\mu_1(l) \wedge \mu_2(l)$.

Example 1: Consider the contract in Figure 4. The program has one global variable `n`, which is initialized to `1` in the constructor. The function `f` can be invoked from the outside of the contract; it increases the value of `n` by 1 every time it is called, but resets it to `1` whenever `n` is `100`. Note that $n \leq 100$ is a transaction invariant: 1) it holds at the end of the constructor, and 2) supposing that $n \leq 100$ holds before entering `f`, we can prove that it also holds when exiting the function. Our algorithm automatically discovers the invariant $n \leq 100$ and succeeds to prove that the assertion at line 5 is safe; upon entering `f`, $n \leq 100$ holds and $n \leq 100 \to n+1 \geq n$ is valid in the theory of unsigned 256 bitvector arithmetic.

Algorithm Structure: Figure 5 describes the overall structure of our algorithm. The input is a smart contract written in Solidity, and the output is a verification result that indicates whether each query (i.e., assertion) in the program is proven safe or not. The algorithm consists of two components, a validator and a generator, where the validator has a solver as a subcomponent.

The algorithm aims to find contract-specific invariants that are inductive and strong enough to prove all provable queries in the given contract. The role of the generator is to produce

candidate invariants that help the validator to prove as many queries as possible. Given a candidate invariant, the validator checks whether the invariant is useful for proving the queries. If it fails to prove the queries, it provides the set of unproven queries as feedback to the generator. The generator uses this feedback to refine the current invariant and generate new ones. This way, the validator and generator form an iterative loop that continuously refines the analysis results until the program is proven to be safe or the given time budget is exhausted. Upon termination, all unproven queries are reported to users as potential safety violations.

Algorithm 1 shows our verification algorithm. It uses a workset (W) to maintain candidate invariants, which initially contains the trivial invariant $(true, \lambda l.true)$ (line 1): the transaction invariant ψ is $true$ and the loop invariant mapping μ maps every label (l) to $true$. The repeat-until loop at lines 2–11 correspond to the feedback loop in Figure 5. At lines 3 and 4, the algorithm chooses and removes a candidate invariant (ψ, μ) from the workset. We choose a candidate invariant that is the smallest in size. At line 5, we run the validator to check whether the current candidate is inductive and strong enough to prove queries, which returns a pair of the boolean variable $inductive$, indicating whether the current candidate invariant is inductive or not, and the set U of unproven queries. If U is empty (line 6), the algorithm terminates and the contract is completely proven to be safe. Otherwise (line 8), we generate a new set of candidate invariants and add them to the workset. Finally, when the current candidate fails to prove some queries but is known to be at least inductive (line 9), we strengthen the remaining candidate invariants using it (line 10), because we can potentially prove more queries with stronger invariants. By doing so, we can find useful invariants more efficiently. The algorithm iterates until it times out or the workset becomes empty. We assume that the algorithm implicitly maintains previously generated invariants to avoid redundant trials.

Technical Contributions: Although the overall algorithm follows the general framework of CEGIS [13], [14], [15], we provide an effective, domain-specific instantiation of the framework in the context of smart contract analysis. Now we describe the details of this instantiation: validator (III-B), generator (III-C), and solver (III-D).

B. Validator

The goal of the validator is to check whether the current candidate invariant (ψ, μ) is inductive and strong enough to prove safety of the queries. The input to the validator is an *annotated program* (c, ψ, μ), i.e., smart contract c annotated with transaction (ψ) and loop (μ) invariants. The validator proceeds in three steps.

Basic Path Construction: Given an annotated program (c, ψ, μ), we first break down the program into a finite set of basic paths [22]. A basic path is a sequence of atomic statements that begins at the entry of a function or a loop, and ends at the exit of a function or the entry of a loop, without passing through other loop entries. We represent a basic path p by the five components: $((l_1, \phi_1), a_1; \ldots; a_n, (l_2, \phi_2))$, where

Algorithm 1 Our Verification Algorithm

Input: A smart contract c to verify
Output: Verification success or potential safety violations

```
1:  W ← {(true, λl.true)}
2:  repeat
3:      Choose a candidate invariant (ψ, μ) from W
4:      W ← W \ {(ψ, μ)}
5:      (inductive, U) ← VALIDATOR(c, ψ, μ)
6:      if U = ∅ then verification succeeds
7:      else
8:          W ← W ∪ GENERATOR(U, ψ, μ)
9:          if inductive then
10:             W ← {(ψ' ∧ ψ, μ' ⋀ μ) | (ψ', μ') ∈ W}
11: until W = ∅ or timeout
12: return potential safety violations
```

l_1 is the label of the starting point (i.e., function or loop entry) of the path, $\phi_1 \in \mathsf{FOL}$ is the invariant annotated at l_1, $a_1, \ldots, a_n$ are atomic statements, l_2 is the label of the end point (i.e., function exit or loop entry) of the path, and $\phi_2 \in \mathsf{FOL}$ is the invariant annotated at l_2. The basic path satisfies the following properties:

1) If l_1 is a function entry, $\phi_1 = \psi$ (i.e., transaction invariant). An exception: $\phi_1 = true$ if l_1 is entry of constructor. If l_2 is a function exit, $\phi_2 = \psi$.
2) Otherwise, i.e., when l_1 and l_2 are labels of loops, $\phi_1 = \mu(l_1)$ and $\phi_2 = \mu(l_2)$ (i.e., considering loop invariants).

Note that our construction of basic paths is exhaustive as we consider *all* paths of the program by summarizing the effects of transactions and loops with their invariants. The basic paths can be computed by traversing control flows of the program.

Example 2: Consider the contract in Figure 4 annotated with the transaction invariant $\psi = n \leq 100$. We do not consider loop invariants as the contract does not have any loops. The annotated program is converted into three basic paths:

$$p_1 : ((entry_0, true), n := 1, (exit_0, n \leq 100))$$
$$p_2 : ((entry_f, n \leq 100), a_1, (exit_f, n \leq 100))$$
$$p_3 : ((entry_f, n \leq 100), a_2, (exit_f, n \leq 100))$$

where $a_1 = assert(n + 1 \geq n); n := n + 1; assume(n \geq 100); n := 1$ and $a_2 = assert(n + 1 \geq n); n := n + 1; assume(n < 100)$. p_1 represents the basic path of the constructor (whose entry and exit labels are $entry_0$ and $exit_0$, respectively). p_2 and p_3 represent the basic paths of the function `f` that follow the true and false branches of the conditional statement at line 7, respectively. Note that conditional statements and loops do not appear as they are broken into basic paths with original conditions given as $assume$ statements.

Generation of Verification Conditions: Let P be the set of basic paths constructed from the annotated program. We next generate verification conditions (VCs) for each basic path.

To derive the VCs, we should be able to express effects of program statements in FOL. To do so, we define a strongest

postcondition predicate transformer $\mathsf{sp} : \mathsf{stmt} \to \mathsf{FOL} \times \mathsf{FOL} \to \mathsf{FOL} \times \mathsf{FOL}$, which is defined in a standard way for each atomic statement as follows:

$$\begin{array}{rcl} \mathsf{sp}(x := e)(\phi_1, \phi_2) & = & (x = e[x'/x] \wedge \phi_1[x'/x], \phi_2) \\ \mathsf{sp}(x[y] := e)(\phi_1, \phi_2) & = & (x = x'\langle y \lhd e[x'/x]\rangle \wedge \phi_1[x'/x], \phi_2) \\ \mathsf{sp}(assume(e))(\phi_1, \phi_2) & = & (\phi_1 \wedge e, \phi_2) \\ \mathsf{sp}(assert(e))(\phi_1, \phi_2) & = & (\phi_1, \phi_2 \wedge (\phi_1 \to e)) \end{array}$$

where unprimed variables (e.g., x) and primed variables (e.g., x') represent the current and previous program states, respectively. In each rule, ϕ_1 is a precondition and sp transforms it into a postcondition while accumulating the safety conditions of assertions in ϕ_2. We write $x'\langle y \lhd e\rangle$ for the modified array x' that stores the value of e at position y. With sp, we define the procedure GENVC that generates the VC of a basic path:

$$\textsc{GenVC}(((l_1, \phi_1), a_1; \ldots; a_n, (l_2, \phi_2))) = (\phi_1' \to \phi_2, \phi_2')$$

where $(\phi_1', \phi_2') = (\mathsf{sp}(a_n) \circ \cdots \circ \mathsf{sp}(a_2) \circ \mathsf{sp}(a_1))(\phi_1, true)$. The generated VC consists of two parts: $\phi_1' \to \phi_2$ is a formula for checking that the annotated invariants are inductive, and ϕ_2' is a formula for checking the safety properties in assertions.

Example 3: Consider the basic path p_3 in Example 2. The corresponding VC is a pair of $(n' \leq 100 \wedge n = n' + 1 \wedge n < 100 \to n \leq 100, n \leq 100 \to n + 1 \geq n)$, both of which are valid in the bitvector theory.

Collecting Unproven Paths: Finally, we return a pair of the boolean variable *inductive* and the subset $U \subseteq P$ of basic paths whose VCs are invalid:

$$(inductive, U) = \begin{cases} \textbf{if } \exists p \in P.\textsc{GenVC}(p).1 \text{ is invalid } \textbf{then} \\ \quad (false, \{p \in P \mid \textsc{GenVC}(p).1 \text{ is invalid}\}) \\ \textbf{else } (true, \{p \in P \mid \exists F \in \textsc{GenVC}(p).2 \text{ is invalid}\}) \end{cases}$$

$\textsc{GenVC}(p).1$ and $\textsc{GenVC}(p).2$ denote the first (i.e., the VC on inductiveness) and the second (i.e., the VC on safety) component of $\textsc{GenVC}(p)$, respectively. We also write $F \in \textsc{GenVC}(p).2$ for a clause of $\textsc{GenVC}(p).2$, where F corresponds to the safety condition of a single query. In the above procedure, we first check whether some VCs regarding inductiveness are invalid. If it does so (if-case), we set *inductive* to *false* and U becomes the basic paths where inductiveness checking failed. Note that, in this case, we accelerate our verification procedure by excluding from U the paths where safety checking may fail. That is, we first focus on refining invariants to be inductive and then strengthen them further to prove safety rather than trying to achieve both at the same time. When the current candidate invariant is inductive (else-case), we set *inductive* to *true* and collect the basic paths where some queries are not proven to be safe. To check the validity of the VCs, we use a domain-specific solver, which will be explained in Section III-D.

C. Generator

The generator takes the set U as feedback and produces new candidate invariants by refining the current one (ψ, μ). $\textsc{Generator}(U, \psi, \mu)$ returns the following set:

$$\{(\psi, \mu') \mid \mu' \in \textsc{Loop}(\mu, U)\} \cup \{(\psi', \mu) \mid \psi' \in \textsc{Tran}(\psi, U)\}$$

where LOOP and TRAN generate new loop and transaction invariants, respectively, based on the current ones. We define $\textsc{Loop}(\mu, U)$ so as to return the following set of refined loop invariants:

$$\bigcup_{((l_1,\_),a,(l_2,\_)) \in U} \{\mu[l_i \mapsto \phi_i] \mid i \in [1,2], \phi_i \in \textsc{RefineL}(\mu(l_i), a)\}$$

where we assume l_1 and l_2 are loop labels, and a is the sequence of atomic statements in the basic path. The definition of $\textsc{Tran}(\psi, U)$:

$$\{\psi' \mid ((l_1, \_), a, (l_2, \_)) \in U, \psi' \in \textsc{RefineT}(\psi, a)\}$$

where we assume l_1 is the label of a function entry or l_2 is the label of a function exit. In the definitions above, the procedures REFINEL and REFINET are actually responsible for refining loop and transaction invariants, which ultimately determine the effectiveness of the generator and the overall verification algorithm.

Domain-Specific Refinement: We define REFINEL and REFINET in terms of *refinement relation*. A refinement relation $(\rightsquigarrow_{X,C}) \subseteq \mathsf{FOL} \times \mathsf{FOL}$ is a binary relation on logical formulas, parameterized by variable set X and constant set C, which describes how a candidate invariant is refined in one step: i.e., ϕ can be refined to any of $\{\phi' \mid \phi \rightsquigarrow_{X,C} \phi'\}$. In our approach, choosing a right refinement relation holds the key to cost-effective verification since it defines the search space of candidate invariants. For example, simply choosing a very general or specific refinement relation would not be practical because of the huge or too limited search space. Instead, we have to carefully design a refinement relation tailored for real-world smart contracts to make our algorithm cost-effective.

Fortunately, we observed that smart contracts in practice share common properties and accordingly considered the following points when we design the refinement relation. First, smart contracts often use loops in simple and restricted forms, e.g., `for(i = 0; i < x ; i++)`, and therefore it is sufficient to consider simple numerical invariants. In particular, we decided to focus on invariants of the forms $x = y$, $x \geq y$, $x = n$, $x \geq n$, and $x \leq n$, where x, y are variables and n denotes integer constants. That is, we do not consider non-linear or compound invariants such as $x = y^2$ and $x = y + z$. Second, because smart contracts use the mapping datatype extensively (e.g., `balance` in token contracts), it is particularly important to capture their common properties (e.g., the sum of `balance` is equal to `totalSupply`). Currently, we support the function symbol sum for variables of mapping type: for example, $\mathsf{sum}(\texttt{balance})$ means the sum of all balances. Third, we consider invariants that are quantifier-free conjunctive formulas. That is, we do not allow disjunctions or quantifiers to be used in candidate invariants.

Based on the observations, we define the refinement relation:

$$\phi_1 \rightsquigarrow_{X,C} \phi_2 \iff \phi_2 = \phi_1 \wedge \varphi \text{ and } \varphi \in A$$

where A is the set of atomic predicates of the forms $x = y, x \geq y, x = n, x \geq n, x \leq n, \mathsf{sum}(x) = e$, where $x, y \in X$, $n \in C$, and $e \in C \cup X$. That is, the current invariant

ϕ_1 is strengthened with a linear and quantifier-free atomic predicate (φ). Note that we only use the symbol sum in the equality predicate as we found invariants of other forms such as $\mathsf{sum}(x) > e$ are rarely used in practice. Finally, we define REFINET and REFINEL using $\rightsquigarrow_{X,C}$ as follows:

$$\begin{array}{rcl} \textsc{RefineL}(\psi, a) & = & \{\psi' \mid \psi \rightsquigarrow_{vars(a),const(a)} \psi'\} \\ \textsc{RefineT}(\phi, a) & = & \{\phi' \mid \phi \rightsquigarrow_{globals,cnstr \cup const(a)} \phi'\} \end{array}$$

where $vars(a)$ and $const(a)$ are the variables and constants appearing in the atomic statements a, respectively. *globals* and *cnstr* represent the set of global variables and constants in the constructor function, respectively. We instantiate the sets X and C differently because transaction invariants often involve global state variables and constants of the entire contract while loop invariants involve local and global variables and constants that appear in the enclosing function. In both cases, we reduce the search space by focusing on local variables and constants to those of the current basic path (a).

D. Solver

The last component is the solver that is used by the validator to discharge the verification conditions. The solver ultimately uses an off-the-shelf SMT solver (we use Z3 [23]) but performs domain-specific preprocessing and optimization steps before using it, which we found important to make our approach practical for real-world contracts. For a basic path p, we assume its verification condition F (either the inductiveness condition, i.e., $F = \textsc{GenVC}(p).1$, or the safety condition of a query, i.e., $F \in \textsc{GenVC}(p).2$) is given.

Preprocessing: Since F may contain symbols (i.e., sum) that conventional SMT solvers cannot understand, we must preprocess F so that all such uninterpretable symbols get replaced by equi-satisfiable formulas in conventional theories. For example, let F contains sum as follows:

$$F = \cdots \wedge \mathsf{sum}(x) = n \wedge x[i] = v_1 \wedge x[j] = v_2 \wedge \cdots$$

where we elide portions of F that are irrelevant to the mapping variable x (i.e., x is only accessed with i and j in the given basic path p). Our idea to translate F into a formula without sum is to instantiate the symbol with respect to the context where F is evaluated. In this example, we can translate the formula F into the following:

$$\cdots \wedge F_1 \wedge F_2 \wedge x[i] = v_1 \wedge x[j] = v_2 \wedge \cdots$$

where $F_1 = (i \neq j \rightarrow x[i] + x[j] + R_x = n) \wedge (i = j \rightarrow x[i] + R_x = n)$ asserts that the sum of distinct elements of x equals n. Because x is used in the given basic path with two index variables i and j, we consider two cases: $i = j$ and $i \neq j$. When $i \neq j$, we replace $\mathsf{sum}(x) = n$ by $x[i] + x[j] + R_x = n$, where R_x is a fresh variable denoting the sum of $x[k]$ for all $k \in domain(x) \setminus \{i, j\}$, where $domain(x)$ is the domain of the mapping. The other case ($i = j$) is handled similarly. F_2 is the additional assertion that guarantees the validity of F_1: $F_2 = (i \neq j \rightarrow x[i] + x[j] \geq x[j] \wedge x[i] + x[j] + R_x \geq R_x) \wedge (i = j \rightarrow x[i] + R_x \geq R_x) \wedge B_x$, where B_x is a fresh propositional variable, meaning that the summations in R_x do not overflow. The general method for our preprocessing is given in Appendix A.

Note that the verification condition after preprocessing can be checked by a conventional SMT solver. However, we found that the resulting formulas are often too complex for modern SMT solvers to handle efficiently, so we apply the following optimization techniques.

Efficient Invalidity Checking: Most importantly, we quickly decide invalidity of formulas without invoking SMT solvers. We observed that even state-of-the-art SMT solvers can be extremely inefficient when our verification conditions are invalid. For example, consider the following formula:

$$true \rightarrow (a - b = 0) \vee (a - b \neq 0 \wedge ((a - b) * 255)/(a - b) = 255).$$

It is easy to see that the formula is invalid in the theory of 256-bit arithmetic (e.g., it does not hold when $a = 2^{255}$ and $b = 0$). Unfortunately, however, the latest version of Z3 [23] (ver 4.8.4) and CVC4 [24] (ver 1.7) takes more than 3 minutes to conclude the formula is invalid.

To mitigate this problem, we designed a simple decision procedure based on the free variables of formulas; given a VC of the form $p \rightarrow q$, we conclude that it is invalid if $\mathsf{FV}(p) \not\supseteq \mathsf{FV}(q)$. The intuition is that p must include more variables than q, as a necessary condition to be stronger than q. In the above example, we conclude the formula is invalid because $\mathsf{FV}(true) \not\supseteq \mathsf{FV}(a = 0 \vee (a \neq 0 \wedge (a * b)/a = b)) = \{a, b\}$. In practice, we found that this simple technique improves the scalability of the verification algorithm significantly as it avoids expensive calls to SMT solvers.

Let us explain why our technique is correct. We first review the notion of interpretation in first-order logic [22]. An interpretation $I : (D_I, \alpha_I)$ is a pair of a domain (D_I) and an assignment (α_I). The domain D_I is a nonempty set of values (or objects). The assignment α_I maps variables, constants, functions, and predicate symbols to elements, functions, and predicates over D_I. Let $J : I \lhd \{x \mapsto v\}$ denote an x-variant of I such that J accords with I on everything except for x. That is, $D_I = D_J$ and $\alpha_I[y] = \alpha_J[y]$ if $y \neq x$, but $\alpha_I[x]$ and $\alpha_J[x]$ may be different. Then, we have the following result (see Appendix B for proof).

Proposition 1: Let p and q be first-order formulas. Then, $p \rightarrow q$ is invalid if the following three conditions hold:

(i) $\mathsf{FV}(p) \not\supseteq \mathsf{FV}(q)$,
(ii) p is satisfiable: $\exists I.\ I \models p$, and
(iii) q has a nontrivial variable: there exists $x \in \mathsf{FV}(q) \setminus \mathsf{FV}(p)$ such that for any interpretation I, if $I \models q$ then $I \lhd \{x \mapsto v\} \models \neg q$ for some $v \in D_I \setminus \{\alpha_I[x]\}$.

Our technique is based on this result but checks the first condition (i) only, which can be done syntactically and efficiently. We do not check the last two conditions (ii) and (iii) as they require invoking SMT solvers in general. Therefore, our technique may decide valid VCs as invalid (i.e., producing false positives) although no invalid VCs are determined to be valid (i.e., no false negatives). Because the technique causes no false negatives, it can be used by sound verifiers.

Although approximated, our technique rarely produces false positives in practice. For example, consider the valid formula $true \rightarrow a \geq a$. Our technique may incorrectly conclude that the formula is invalid, since $\mathsf{FV}(true) \not\supseteq \mathsf{FV}(a \geq a)$ but we do not check the condition (iii) that the formula violates. Note that, however, such a *trivial* formula is unlikely to appear during the verification of real-world smart contracts; the verification condition $true \rightarrow a \geq a$ would be generated from the trivial expression $a-a$ that does not appear frequently in programs. Even when they appear, we can easily remove the *triviality*. For example, it is easy to simplify $true \rightarrow a \geq a$ into $true \rightarrow true$ that is not determined as invalid by our technique since $\mathsf{FV}(true) \supseteq \mathsf{FV}(true)$. In fact, no false positives were caused by our technique in our experiments in Section V.

Efficient Validity Checking: We also quickly identify some valid formulas by using a number of domain-specific templates. This is because our verification conditions are likely to involve arrays and non-linear expressions extensively but modern SMT solvers are particularly inefficient for handling them. For example, a simple yet important validity template is as follows:

$$\frac{}{F' \rightarrow x \geq (x * n_1)/n_2} \; n_1 \leq n_2$$

where F' denotes an arbitrary formula, x a 256-bit unsigned integer variable, and n_1 and n_2 some integer constants. This template asserts that, regardless of the precondition F', $x \geq (x * n_1)/n_2$ holds if $n_1 \leq n_2$. Using the template, we can conclude that a formula $\cdots \rightarrow y \geq (y * 99)/100$ is valid (i.e., the subtraction $y - (y * 99)/100$ is safe from underflow) without calling an external SMT solver. These templates are used before the preprocessing step; several templates were designed to determine the validity of formulas containing domain-specific symbols at a high level without preprocessing. We provide more examples in Appendix C.

IV. Implementation

In this section, we explain implementation details of VeriSmart, which consists of about 7,000 lines of OCaml code. Although Section III describes our algorithm for a small subset of Solidity, our implementation supports the full language (except for inline assembly). Most Solidity features (e.g., function modifers) can be desugared into our core language in a straightforward way. We discuss nontrivial issues below.

Function Calls: Basically, we handle function calls by inlining them into their call-sites up to a predefined inlining depth k (currently, less than or equal to 2). Exceptions include relatively large functions (with more than 20 statements) that might cause scalability issues and inter-contract function calls (i.e., calling functions in other contracts via contract objects). To perform exhaustive verification, we handle those remaining function calls conservatively as follows.

First, we conservatively reflect side-effects of function calls on the caller side. To do so, we first run a side-effect analysis [25] to find variables whose values may be changed by the called functions. Next, we weaken the formulas at call-sites by replacing each of atomic predicates that involve those variables by *true*. For example, consider a call statement `x:=foo()` and assume `foo` may change the value of variable `a` in its body. Suppose further the precondition of the call-site is $a \geq 1 \wedge b \geq 1 \wedge c \geq 1 \wedge x \geq y$. Then, we obtain the following postcondition of the call-site: $true \wedge b \geq 1 \wedge c \geq 1 \wedge true$ where $a \geq 1$ and $x \geq y$ get replaced by $true$. Regarding inter-contract function calls, it is enough to invalidate the value of return variables only, as inter-contract calls in Solidity cannot directly modify other contracts' states. For example, consider the precondition above and an inter-contract call `x := o.foo()`. We produce the postcondition $a \geq 1 \wedge b \geq 1 \wedge c \geq 1 \wedge true$, where only $x \geq y$ is replaced by $true$.

Second, we separately analyze function bodies not inlined. This step is needed to detect potential bugs in the functions skipped during the step described in the preceding paragraph. To perform exhaustive verification, we analyze these functions by over-approximating their input states. Specifically, when the function in a main contract has `public` or `external` visibility, we run the algorithm in Section III which annotates entry and exit with transaction invariant. On the other hand, when the function in a main contract has `internal` or `private` visibility (i.e., the functions which cannot be called from the outside and can only be accessed via function call statements) or the function is defined in other contracts, we generate the VCs after we annotate entries and exits of them with *true*, i.e., incoming state at the entry is over-approximated as *true* and inductiveness condition can be trivially checked at the exit.

In summary, VeriSmart performs exhaustive safety verification without missing any possible behaviors. In theory, we may lose precision due to the conservative function-call analysis. However, as our experimental results in Section V demonstrate, our approach is precise enough in practice.

Inheritance: In Section III, we assumed a single contract is given. To support contract inheritance, we copy functions and global variables of parent contracts to a main contract using the inheritance graph provided by the Solidity compiler. During this conversion, we consider function overriding and variable hiding, and do not copy functions with the same signatures and the same variables.

Structures: We encode structures in Solidity with arrays. To do so, we introduce a special mapping variable for each member of a structure type, which maps structures to the member values. For example, given a precondition ϕ, the strongest postcondition of command `x.y := z` is $m_y = m'_y\langle x \lhd z\rangle \wedge \phi[m'_y/m_y]$, where m_y is a map (or an array) from structures to the corresponding values of member `y` and x is an uninterpreted symbol for the structure variable `x`. Note that we are able to handle aliasing among structures using this encoding. For example, if two structures `p` and `q` are aliased and they both have `y` as a member, then we can access the same member `y` using either of the structures, i.e., $m_y[p] = m_y[q]$.

Inline Assembly: One potential source of false negatives of source code analyzer (e.g., Zeus [11]) is inline assembly.

VERISMART also has this limitation and may miss bugs hidden in embedded bytecode. However, VERISMART conservatively analyzes the remaining parts of the source code by considering the side-effects of the assembly blocks in a similar way that we handle function call statements, i.e., we replace each atomic predicate by *true* if it involves variables used in assembly code (using the information provided by the Solidity compiler). Note that this limitation does not impair the practicality of VERISMART significantly, as inline assembly is not very common in practice. For example, in our benchmarks in Section V, only four contracts (#4, #16, #52 in Table II, #24 in Table IV) contain assembly blocks but none of these assembly blocks include arithmetic operations.

V. EVALUATION

We evaluate the effectiveness of VERISMART by comparing it with existing tools. Research questions are as follows:

(1) How precisely can VERISMART detect arithmetic bugs compared to the existing bug-finders, i.e., OSIRIS [7], OYENTE [9], MYTHRIL [8], MANTICORE [10]?
(2) How does VERISMART compare to the existing verifiers, i.e., ZEUS [11] and SMTCHECKER [12]?

In addition, we conduct a case study to show VERISMART can be easily extended to support other types of vulnerabilities (Section V-C). We used the latest versions of the existing tools (as of May 1st, 2019). All experiments were conducted on a machine with Intel Core i7-9700K and 64GB RAM.

A. Comparison with Bug-finders

We evaluate the bug-finding capability of VERISMART by comparing it with four bug-finding analyzers for Ethereum smart contracts: OSIRIS [7], OYENTE [26], MYTHRIL [8], and MANTICORE [10]. They are well-known open-sourced tools that support detection of integer overflows (OSIRIS, OYENTE, MYTHRIL, MANTICORE) and division-by-zeros (MYTHRIL). In particular, OSIRIS is arguably the state-of-the-art tailored for finding integer overflow bugs [7].

Setup: We used 60 smart contracts that have vulnerabilities with assigned CVE IDs. We have chosen these contracts to enable in-depth manual study on the analysis results with known vulnerabilities confirmed by CVE reports. The 60 benchmark contracts were selected randomly from the 487 CVE reports that are related to arithmetic overflows (Table I), excluding duplicated contracts with minor syntactic differences (e.g., differences in contract names or logging events). During evaluation, we found four incorrect CVE reports (#13, #20, #31, #32 in Table II), which will be discussed in more detail at the end of the section.

To run OSIRIS, OYENTE, MYTHRIL, and MANTICORE, we used public docker images provided together with these tools. Following prior work [7], we set the timeout to 30 minutes per contract. For fair comparison, we activated only the analysis modules for arithmetic bug detection when such option is available (MYTHRIL, MANTICORE). We left other options as default. For VERISMART, we set the timeout to 1 minute for the last entrance of the loop in Algorithm 1, and set the timeout to 10 seconds for Z3 request, because these numbers worked effectively in our experience; if we set each timeout to a lower value, the precision may decrease (Section V-D). In analysis reports of each tool, we only counted alarms related to arithmetic bugs (integer over/underflows and division-by-zeros) for a main contract whose name is available at the Etherscan website [27].

Results: Table II shows the evaluation results on the CVE dataset. For each benchmark contract and tool, the table shows the number of alarms (#Alarm) and the number of false positives (#FP) reported by the tool; regarding these two numbers, we did not count cases where the tools (OYENTE and MYTHRIL) ambiguously report that the entire body of a function or the entire contract is vulnerable. The CVE columns indicate whether the tool detected the vulnerabilities in CVE reports or not (✓: a tool successfully pinpoints all vulnerable locations in each CVE report, ✗: a tool does not detect any of them, △: a tool detects only a part of vulnerable points in each CVE report or, obscurely reports the body of an entire function containing CVE vulnerabilities is vulnerable without pinpointing specific locations. N/A: all vulnerabilities in CVE reports are actually safe; see Table III).

The results show that VERISMART far outperforms the existing bug-finders in both precision and recall. In total, VERISMART reported 492 arithmetic over/underflow and division-by-zero alarms. We carefully inspected these alarms and confirmed that 490 out of 492 were true positives (i.e., safety can be violated for some feasible inputs), resulting in a false positive rate ($\frac{\#FP}{\#Alarm}$) of 0.41% (2/492). We also inspected 484 (=976-492) unreported queries to confirm that all of them are true negatives (i.e., no feasible inputs exist to violate safety), resulting in a recall of 100%. Of course, VERISMART detected all CVE vulnerabilities. In contrast, existing bug-finders missed many vulnerabilities. For example, OSIRIS managed to detect 41 CVE vulnerabilities with 17 undetected known vulnerabilities. OYENTE pinpointed 20 exact vulnerable locations in CVE, partly detected vulnerabilities in 4 CVE reports, vaguely raised alarms on 11 functions containing vulnerable locations, and missed 23 CVE vulnerabilities. MYTHRIL detected vulnerabilities in 10 CVE reports, obscurely warned that 1 function is vulnerable, and missed 46 known issues. MANTICORE was successful in only two CVE reports, failing on 42 CVE reports. The false positive rates of OSIRIS, OYENTE, and MYTHRIL were 5.42% (13/240), 8.19% (14/171), and 10.64% (10/94), respectively.

Efficiency: VERISMART was also competitive in terms of efficiency. To obtain the results in Table II on the 60 benchmark programs, VERISMART, OSIRIS, OYENTE, MYTHRIL, and MANTICORE took 1.1 hour (3,807 seconds), 4.2 hours (14,942 seconds), 14 minutes, 13.8 hours (49,680 seconds), and 31.4 hours (112,920 seconds) respectively, excluding the cases of timeout (though we set the timeout to 30 minutes, MANTICORE sometimes did not terminate within 3 days) and internal errors (e.g., unsupported operations encountered, abnormal termination) of MYTHRIL and MANTICORE.

TABLE II

EVALUATION OF EXISTING TOOLS ON CVE REPORTS. LOC: LINES OF CODE. #Q: THE TOTAL NUMBER OF QUERIES FOR EACH CONTRACT AFTER REMOVING UNREACHABLE FUNCTIONS. #ALARM: THE NUMBER OF ENTIRE ALARMS PRODUCED BY EACH TOOL. #FP: THE NUMBER OF FALSE ALARMS. CVE: A MARKER THAT INDICATES WHETHER EACH TOOL SUCCESSFULLY DETECTS VULNERABILITIES IN CVE. ✓: A TOOL SUCCESSFULLY PINPOINTS ALL VULNERABLE LOCATIONS IN CVE. △: A TOOL DETECTS ONLY A PART OF VULNERABILITIES IN CVE, OR OBSCURELY REPORTS THAT AN ENTIRE FUNCTION BODY IS VULNERABLE WITHOUT PINPOINTING SPECIFIC LOCATIONS. ✗: A TOOL TOTALLY FAILED TO DETECT VULNERABILITIES IN CVE. N/A: ALL VULNERABILITIES REPORTED IN CVE ARE ACTUALLY SAFE (#13, #31). FOR PARTLY CORRECT CVE REPORTS (#20, #32), THE CVE INFORMATION IS VALID W.R.T. THEM.

No.	CVE ID	Name	LOC	#Q	VERISMART			OSIRIS [7]			OYENTE [9], [26]			MYTHRIL [8]			MANTICORE [10]		
					#Alarm	#FP	CVE	#Alarm	#FP	CVE	#Alarm	#FP	CVE	#Alarm	#FP	CVE	#Alarm	#FP	CVE
#1	2018-10299	BEC	299	6	2	0	✓	0	0	✗	1	0	△	2	0	✓	0	0	✗
#2	2018-10376	SMT	294	22	13	0	✓	1	0	✓	2	0	✗	1	0	✗	timeout (> 3 days)		
#3	2018-10468	UET	146	27	14	0	✓	9	0	✗	8	0	✓	5	0	✓	0	0	✗
#4	2018-10706	SCA	404	48	33	0	✓	9	0	✗	4	0	△	2	0	✗	internal error		
#5	2018-11239	HXG	102	11	7	0	✓	6	0	✓	2	0	✗	3	0	✓	2	0	✓
#6	2018-11411	DimonCoin	126	15	7	0	✓	5	0	✗	5	0	✓	5	0	✓	3	0	✓
#7	2018-11429	ATL	165	9	4	0	✓	3	0	✓	2	0	△	0	0	✗	0	0	✗
#8	2018-11446	GRX	434	39	24	2	✓	8	2	✗	12	4	✗	4	2	✗	internal error		
#9	2018-11561	EETHER	146	10	5	0	✓	4	0	✓	2	0	△	2	0	✓	0	0	✗
#10	2018-11687	BTCR	99	20	4	0	✓	2	0	✓	2	0	△	3	2	✗	0	0	✗
#11	2018-12070	SEC	269	40	8	0	✓	6	0	✓	4	0	✗	3	1	✗	0	0	✗
#12	2018-12230	RMC	161	9	5	0	✓	3	0	✓	5	0	✓	0	0	✗	0	0	✗
#13	2018-13113	ETT	142	9	2	0	N/A	4	2	N/A	2	2	N/A	0	0	N/A	0	0	N/A
#14	2018-13126	MoxyOnePresale	301	5	3	0	✓	0	0	✗	0	0	✗	0	0	✗	0	0	✗
#15	2018-13127	DSPX	238	6	4	0	✓	3	0	✓	3	0	△	1	0	✗	0	0	✗
#16	2018-13128	ETY	193	10	4	0	✓	3	0	✓	3	0	△	0	0	✗	0	0	✗
#17	2018-13129	SPX	276	9	6	0	✓	5	0	✓	3	0	△	1	0	✗	internal error		
#18	2018-13131	SpadePreSale	312	4	3	0	✓	0	0	✗	0	0	✗	0	0	✗	internal error		
#19	2018-13132	SpadeIco	403	9	6	0	✓	0	0	✗	0	0	✗	0	0	✗	internal error		
#20	2018-13144	PDX	103	5	2	0	✓	2	1	✓	2	1	✓	internal error			0	0	✗
#21	2018-13189	UNLB	335	4	3	0	✓	2	0	✓	3	0	✓	1	0	✗	0	0	✗
#22	2018-13202	MyBO	183	17	11	0	✓	5	0	✓	3	0	✗	1	0	✗	internal error		
#23	2018-13208	MoneyTree	171	17	10	0	✓	4	0	✓	2	0	✗	2	0	✗	0	0	✗
#24	2018-13220	MAVCash	171	15	10	0	✓	4	0	✓	2	0	✗	1	0	✗	0	0	✗
#25	2018-13221	XT	186	15	10	0	✓	4	0	✓	2	0	✗	2	0	✗	0	0	✗
#26	2018-13225	MyYLCToken	181	17	11	0	✓	5	0	✓	6	0	✗	0	0	✗	0	0	✗
#27	2018-13227	MCN	172	17	10	0	✓	4	0	✓	2	0	✗	2	0	✗	0	0	✗
#28	2018-13228	CNX	171	17	10	0	✓	4	0	✓	2	0	✗	2	0	✗	0	0	✗
#29	2018-13230	DSN	171	17	10	0	✓	4	0	✓	2	0	✗	2	0	✗	0	0	✗
#30	2018-13325	GROW	176	12	2	0	✓	4	2	✓	1	1	✗	0	0	✗	0	0	✗
#31	2018-13326	BTX	135	9	2	0	N/A	4	2	N/A	2	2	N/A	0	0	N/A	0	0	N/A
#32	2018-13327	CCLAG	92	5	2	0	✓	2	1	✓	2	1	✓	0	0	✗	0	0	✗
#33	2018-13493	DaddyToken	344	40	22	0	✓	8	0	✗	2	0	✗	3	0	✗	internal error		
#34	2018-13533	ALUXToken	191	23	13	0	✓	8	0	✓	2	0	✓	1	0	✗	1	0	✗
#35	2018-13625	Krown	271	22	9	0	✓	1	0	✗	3	0	✓	0	0	✗	internal error		
#36	2018-13670	GFCB	103	14	11	0	✓	6	1	✓	3	1	✓	1	0	✗	0	0	✗
#37	2018-13695	CTest7	301	17	8	0	✓	0	0	✗	0	0	✗	0	0	✗	0	0	✗
#38	2018-13698	Play2LivePromo	131	8	7	0	✓	7	0	✓	7	0	✓	5	0	✗	5	0	✗
#39	2018-13703	CERB_Coin	262	17	8	0	✓	5	0	✓	2	0	✗	2	1	✗	0	0	✗
#40	2018-13722	HYIPToken	410	8	3	0	✓	2	0	✓	2	0	✓	0	0	✗	internal error		
#41	2018-13777	RRToken	166	8	3	0	✓	2	0	✓	2	0	✓	0	0	✗	0	0	✗
#42	2018-13778	CGCToken	224	13	6	0	✓	4	0	✓	4	0	✓	1	0	✗	1	0	✗
#43	2018-13779	YLCToken	180	17	11	0	✓	5	0	✓	6	0	✓	0	0	✗	0	0	✗
#44	2018-13782	ENTR	171	17	10	0	✓	4	0	✓	2	0	✓	2	0	✗	0	0	✗
#45	2018-13783	JiucaiToken	271	19	11	0	✓	6	0	✓	4	0	✓	0	0	✗	internal error		
#46	2018-13836	XRC	119	22	7	0	✓	5	0	✗	3	0	△	3	1	✓	timeout (> 3 days)		
#47	2018-14001	SKT	152	19	10	0	✓	4	0	✗	3	0	△	3	0	✓	0	0	✗
#48	2018-14002	MP3	83	12	4	0	✓	2	0	✗	2	0	△	2	1	✗	timeout (> 3 days)		
#49	2018-14003	WMC	200	15	6	0	✓	3	0	✗	2	0	△	3	0	✓	1	0	✗
#50	2018-14004	GLB	299	40	8	0	✓	5	0	✓	1	0	△	0	0	✗	0	0	✗
#51	2018-14005	Xmc	255	29	11	0	✓	8	0	✓	1	0	△	3	0	△	0	0	✗
#52	2018-14006	NGT	249	27	13	0	✓	1	0	✗	5	0	△	0	0	✗	timeout (> 3 days)		
#53	2018-14063	TRCT	178	9	1	0	✓	1	0	✓	1	0	✓	4	2	✓	0	0	✗
#54	2018-14084	MKCB	273	17	10	0	✓	5	0	✓	4	0	✗	2	0	✗	1	0	✗
#55	2018-14086	SCO	107	16	14	0	✓	7	2	✓	5	2	✗	0	0	✗	0	0	✗
#56	2018-14087	EUC	174	15	7	0	✓	4	0	✗	4	0	✗	0	0	✗	0	0	✗
#57	2018-14089	Virgo_ZodiacToken	208	30	20	0	✓	12	0	✓	5	0	✓	14	0	✓	0	0	✗
#58	2018-14576	SunContract	194	12	4	0	✓	1	0	✓	0	0	✗	0	0	✗	0	0	✗
#59	2018-17050	AI	141	8	3	0	✓	1	0	✓	1	0	✓	0	0	✗	0	0	✗
#60	2018-18665	NXX	79	7	5	0	✓	4	0	✓	4	0	✓	0	0	✗	0	0	✗
Total			12493	976	492	2	✓:58 △: 0 ✗ : 0	240	13	✓:41 △: 0 ✗ :17	171	14	✓:20 △:15 ✗ :23	94	10	✓:10 △: 1 ✗ :46	14	0	✓: 2 △: 0 ✗ :42

False Alarms of Bug-finders: To see why VERISMART achieves higher precision than bug-finders, we inspected all 37 (=13+14+10) false positives reported by bug-finders. Bug-finders reported 18 among 37 false positives due to the lack of inferring transaction invariants, all of which are avoided by VERISMART. The remaining 19 false positives were due to imprecise handling of conditional statements. For example, consider the following code snippet (from #55):

```
function transfer(address _to, uint _value) {
  if (msg.sender.balance < min)
    sell((min - msg.sender.balance) / sellPrice);
}
```

where the safety of `min - msg.sender.balance` is ensured by the preceding guard. Both OSIRIS and OYENTE incorrectly reported that the subtraction is unsafe and integer underflow would occur. This might be because OSIRIS and OYENTE do not keep track of complex path conditions (e.g., involving structures in this case) for some engineering issues. In contrast, VERISMART analyzes every conditional statement

```
1  function unlockReward(address addr, uint value) {
2    require(totalLocked[addr] > value);
3    require(locked[addr][msg.sender] >= value);
4    if(value == 0) value = locked[addr][msg.sender];
5    totalLocked[addr] -= value;  // false positive
6    locked[addr][msg.sender] -= value;
7  }
```

Fig. 6. A function simplified from the benchmark #8. OSIRIS, OYENTE, and VERISMART warn that the subtraction at line 5 can cause arithmetic underflow, which is false positive (i.e., the subtraction is safe).

precisely and do not produce such false alarms.

***False Alarms of* VERISMART:** VERISMART produced two false alarms in the benchmark #8, because it is currently unable to capture quantified transaction invariants. Consider the `unlockReward` function in Figure 6. The subtraction operation at line 5 seems to cause arithmetic underflow; the `value` may be changed at line 4, and thereafter the relation `totalLocked[addr] > value` seems not to hold anymore. However, the subtraction is safe because the following transaction invariant holds over the entire contract:

$$\forall x.\texttt{totalLocked[x]} = \sum_i \texttt{locked[x][}i\texttt{]} \quad (2)$$

with an additional condition that computing the summation ($\sum_i$ `locked[x][`i`]`) does not cause overflow. With this transaction invariant, `value` is always less than `totalLocked[addr]`. Because VERISMART considers quantifier-free invariants only (Section III-C), it falsely reported that an underflow would occur at line 5. OSIRIS and OYENTE produced the false alarm too at the same location.

***False Negatives of Bug-finders*:** We inspected CVE vulnerabilities that were commonly missed by the four bug-finders, and we found that the bug-finders often fail to detect bugs when vulnerabilities could happen via inter-contract function calls. For example, consider code adapted from #18:

```
function mint (address holder, uint value) {
  require (total+ value <= TOKEN_LIMIT); // CVE bug
  balances[holder] += value;             // CVE bug
  total += value;                        // CVE bug
}
```

There is a function call `token.mint (...,...)` in a main contract, where `token` is a contract object. We can see that all three addition operations possibly overflow with some inputs. For example, suppose `total=1`, `value=0xfff...ff`, and `TOKEN_LIMIT=10000`. Then, `total+value` overflows in unsigned 256-bit and thus the safety checking statement can be bypassed. Next, if `balances[holder]=0`, the `holder` can have tokens more than the predetermined limit `TOKEN_LIMIT`. VERISMART detected the bugs as it conservatively analyzes inter-contract calls (Section IV).

***Incorrect CVE Reports Found by* VERISMART:** Interestingly, VERISMART unexpectedly identified six incorrectly-reported CVE vulnerabilities. In Table III, the column # Incorrect Queries denotes the number of queries incorrectly reported to be vulnerable for each CVE ID. We could discover them as VERISMART did not produce any alarms for those queries and then we manually confirmed that the CVE reports are actually incorrect. We have submitted a request for revising these issues to the CVE assignment team.

TABLE III
LIST OF INCORRECT CVE REPORTS FOUND BY VERISMART. #INCORRECT QUERIES: THE NUMBER OF INCORRECTLY REPORTED QUERIES TO BE VULNERABLE. #FP: THE NUMBER OF ALARMS RAISED BY EACH TOOL FOR THE INCORRECTLY REPORTED QUERIES.

CVE ID	Name	#Incorrect Queries	#FP		
			OSIRIS	OYENTE	VERISMART
2018-13113	ETT	2	2	2	0
2018-13144	PDX	1	1	1	0
2018-13326	BTX	2	2	2	0
2018-13327	CCLAG	1	1	1	0

With the capability of automatically computing transaction invariants, VERISMART successfully proved the safety for all the incorrectly reported vulnerabilities (i.e., zero false positives). In other words, VERISMART could not have discovered incorrect CVE reports if it were without transaction invariants. The transaction invariants generated for proving the safety were similar to those in Example 3 of Section II. In contrast, existing bug-finders cannot be used for this purpose such as proving the safety; for example, OSIRIS and OYENTE produced false positives for *all* of the 6 safe queries (i.e., the 6 incorrectly reported queries).

B. Comparison with Verifiers

We now compare VERISMART with SMTCHECKER [12] and ZEUS [11], two recently-developed verifiers for smart contracts. In particular, SMTCHECKER is the "official" verifier for Ethereum smart contracts developed by the Ethereum Foundation, which is available in the Solidity compiler. Like VERISMART, the primary goal of SMTCHECKER is to detect arithmetic over/underflows and division-by-zeros [12].

***Setup*:** First of all, we must admit that the comparison with ZEUS and SMTCHECKER in this subsection is rather limited, because ZEUS is not publicly available and SMTCHECKER is currently an experimental tool that does not support the full Solidity language. Since we cannot run ZEUS on our dataset, the only option was to use the public evaluation data [28] provided by the ZEUS authors. However, the public data was not detailed enough to accurately interprete as the ZEUS authors classify each benchmark contract simply as 'safe' or 'unsafe' without specific alarm information such as line numbers. The only objective information we could obtain from the data [28] was the fact that ZEUS produces some (nonzero) number of false (arithmetic-overflow) alarms on 40 contracts, and we decided to use those in our evaluation. Starting with those 40 contracts, we removed duplicates with trivial syntactic differences, resulting in a total of 25 unique contracts (Table IV). Thus, the objective of our evaluation is to run VERISMART and SMTCHECKER on the 25 contracts to see how many of them can be successfully analyzed by VERISMART and SMTCHECKER without false alarms. We ran SMTCHECKER with the default setting.

TABLE IV
EVALUATION ON THE ZEUS DATASET. VERIFIED: A TOOL DETECTS ALL BUGS WITHOUT FALSE POSITIVES (✓: SUCCESS, ✗: FAILURE)

No.	LOC	#Q	VERISMART			SMTCHECKER [12]			ZEUS [11]
			#Alarm	#FP	Verified	#Alarm	#FP	Verified	Verified
#1	42	3	0	0	✓	3	3	✗	✗
#2	78	2	1	0	✓	2	1	✗	✗
#3	75	7	2	0	✓	7	5	✗	✗
#4	70	7	0	0	✓	7	7	✗	✗
#5	103	8	0	0	✓	6	6	✗	✗
#6	141	5	2	0	✓	internal error			✗
#7	74	6	1	0	✓	6	5	✗	✗
#8	84	6	0	0	✓	4	4	✗	✗
#9	82	6	0	0	✓	6	6	✗	✗
#10	99	2	1	0	✓	internal error			✗
#11	171	15	9	0	✓	internal error			✗
#12	139	7	0	0	✓	internal error			✗
#13	139	7	0	0	✓	internal error			✗
#14	139	7	0	0	✓	internal error			✗
#15	139	7	0	0	✓	internal error			✗
#16	141	16	10	0	✓	internal error			✗
#17	153	5	0	0	✓	internal error			✗
#18	139	7	0	0	✓	internal error			✗
#19	113	4	0	0	✓	4	4	✗	✗
#20	40	3	0	0	✓	3	3	✗	✗
#21	59	3	0	0	✓	internal error			✗
#22	28	3	1	0	✓	1	0	✓	✗
#23	19	3	0	0	✓	3	3	✗	✗
#24	457	30	13	6	✗	internal error			✗
#25	17	3	0	0	✓	3	3	✗	✗
Total	2741	172	40	6	✓:24 ✗ : 1	55	50	✓: 1 ✗: 12	✓: 0 ✗:25

Results: Table IV shows the evaluation results on the ZEUS dataset. For each contract, the table shows the number of alarms (#Alarm), the number of false positives (#FP) produced by VERISMART and SMTCHECKER. The column Verified indicates whether each tool detected all bugs without false positives (✓: success, ✗: failure).

The results show that VERISMART successfully addresses limitations of ZEUS and SMTCHECKER. The 25 contracts contain 172 arithmetic operations, where VERISMART pointed out 40 operations as potential bugs. We have manually checked that 34 out of total alarms are true positives. In benchmark #24, VERISMART produced 6 false positives due to unsupported invariants (quantified invariants and compound invariants, Section III-C), and imprecise function call analysis. We manually checked that the remaining 132 (=172-40) queries proven to be safe by VERISMART are actually true negatives. By contrast, according to the publicly available data [28], ZEUS produces at least one false positives for each contract in Table IV (i.e., $\geq$ 25 false alarms in total). SMTCHECKER could only analyze 13 contracts as it raised internal errors for the other 12 contracts, which is due to its immature support of Solidity syntax [29]. Among 61 operations from 13 contracts, SMTCHECKER succeeded to detect all 5 bugs in them thanks to its exhaustive verification approach. However, it reported 55 alarms in total, of which 50 are false positives. In terms of efficiency, SMTCHECKER took about 1 second per contract and VERISMART took about 20 seconds per contract.

Importance of Transaction Invariants: The key enabler for high precision was the ability of VERISMART to leverage transaction invariants. We also ran VERISMART without inferring transaction invariants (i.e., using *true* as transaction invariants); without transaction invariants, VERISMART fails to verify 17 out of 25 contracts.

C. Case Study: Application to Other Types of Vulnerabilities

VERISMART can be used for analyzing other safety properties as well. To show this, we applied VERISMART to finding bugs related to access control, where security-sensitive variables can be manipulated by anyone for malicious use. For example, consider the code snippet adapted from the EtherCartel contract for crypto idle game (CVE 2018-11329):

```
function DrugDealer() public { ceoAddr = msg.sender; }
function buyDrugs () public payable {
  ceoAddr.transfer(msg.value); // send Ether to ceoAddr
  drugs[msg.sender] += ...; // buy drugs by paying Ether
}
```

Observe that the address-typed variable `ceoAddr`, the beneficiary of Ether, can be taken by anyone who calls the function `DrugDealer`. If an attacker becomes the beneficiary by calling `DrugDealer`, the attacker might illegally take some digital assets whenever benign users buy some digital assets (i.e., drugs) by calling `buyDrugs` where `transfer` in it is a built-in function that sends Ether to `ceoAddr`. This vulnerability was exploited in about 1 hour after deployment [30].

To detect this bug, we used VERISMART as follows. First, we specified safety properties by automatically generating the assertion `assert(msg.sender==addr)` right before each assignment of the form `addr=...;`, where `addr` is a global address-typed variable which is often security-sensitive (excluding assignments in constructors, which typically set the contract owners). Next, we ran VERISMART without any modification of its verification algorithm. With this simple extension, VERISMART worked effectively; it not only detected all known CVE vulnerabilities (2018-10666, 2018-10705, 2018-11329) but also proved the absence of this bug scenario for 55 contracts out of 60 from Table II. VERISMART could not prove safety of the remaining 5 contracts due to the imprecise specification described above.

D. Threats to Validity

We summarize limitations of our evaluation and consequent threats to validity. Firstly, the benchmark contracts that we used (60 CVE dataset + 25 ZEUS dataset) might not be representative although we made effort to avoid bias in the datasets (e.g., removal of duplicates). Secondly, the performance of VERISMART may vary depending on the performance of the off-the-shelf SMT solver (i.e., Z3) used internally or timeout options used in the experiments. Thirdly, we did not study the exploitability of bugs in this paper and did not compare VERISMART and other tools in this regard. Thus, the results may be different if those tools are evaluated with exploitability in mind. Lastly, although we did our best, we realized that manually classifying static analysis alarms into true or false positives is extremely challenging and the classification can be even subjective in a few cases.

VI. RELATED WORK

In this section, we place our work in the literature and clarify our contributions regarding existing works. Section VI-A

compares our work with existing smart contract analyses. Section VI-B discusses verification techniques for other domains.

A. Analyzing Smart Contracts

Compared to existing techniques for analyzing smart contracts [9], [26], [8], [18], [7], [31], [32], [33], [34], [12], [11], [19], [20], [35], [36], [37], [38], [39], [40], VERISMART is unique in that it achieves full automation, high precision, and high recall at the same time. Below, we classify existing approaches into fully automated and semi-automated approaches.

Fully Automated Approaches: VERISMART belongs to the class of fully automated tools based on static or dynamic program analysis techniques that require no manual effort and can be used by end-users who lack expertise in formal verification. Instead, these approaches focus on relatively simple safety properties (e.g., overflows).

One popular approach is bug-finders based on symbolic execution or fuzz testing. For example, OYENTE [9], [26], MYTHRIL [8], OSIRIS [7], MANTICORE [10] and MAIAN [18] discover bugs by symbolically executing EVM bytecode. OYENTE is the first such tool for Ethereum smart contracts, which detects various bug patterns including arithmetic bugs. MYTHRIL is also a well-known open-sourced tool for detecting a variety of bugs by performing symbolic execution. OSIRIS [7] is a tool that is specially designed for detecting arithmetic bugs. MAIAN [18] focuses on finding violations of trace properties. GASPER [31] uses symbolic execution to identify gas-costly programming patterns. REGUARD [34] and ContractFuzzer [41] use fuzz testing to detect common security vulnerabilities. Although symbolic execution and fuzz testing are effective for finding bugs, they inevitably miss critical vulnerabilities, which is particularly undesirable for safety-critical software like smart contracts.

Other approaches are verifiers that perform exhaustive analyses based on static analysis or automatic program verification techniques. ZEUS [11] is a sound static analyzer that can detect arithmetic bugs or prove their absence. ZEUS leverages abstract interpretation and software model checking [42]. SMTCHECKER [12] is the "official" verifier for Solidity developed by the Ehtereum Foundation. Its primarily goal is to verify the absence of arithmetic bugs such as integer over/underflows and division-by-zeros [12] by performing SMT-based bounded verification. Unlike VERISMART, ZEUS and SMTCHECKER lack inter-transactional reasoning and this is currently considered a key limitation of these tools [11], [12].

SECURIFY [19], MadMax [20], and Vandal [21] use declarative static analysis techniques based on Datalog [43]. Besides their inability to infer transaction invariants, one common drawback of Datalog-based analyzers is that they cannot describe general classes of (in particular, numerical) static analyses and is inappropriate for finding arithmetic bugs.

Semi-Automated Approaches: Semi-automated tools for formally specifying and verifying smart contracts have different goals. These approaches can prove a wide range of functional properties at the expense of full automation; they require users to manually provide specifications or invariants.

Hirai [36] formalizes the Ethereum Virtual Machine (EVM) and provides a way to prove safety properties of smart contracts in interactive theorem provers such as Isabelle/HOL [44]. Bharagavan et al. [37] provide a framework for formally specifying and verifying functional correctness of smart contracts using the F* proof assistant [45]. Grishchenko et al. [38] also use F* to formalize small-step semantics of EVM bytecode and express a number of security properties of smart contracts. Hildenbrandt et al. [46] define formal semantics of EVM using the K framework [47]. Amani et al. [39] formalize EVM in Isabelle/HOL and provide a program logic for reasoning about smart contracts. Lahiri et al. [40] describe an approach for formal specification and verification of smart contracts, where the primary goal is to take a high-level specification expressed by a state machine and to verify that the implementation meets the specification.

Manual Safety Checking: Some techniques (e.g., SafeMath [48]) depend on manual annotation of programs to prevent bugs, which has two drawbacks. First, manual annotation is error-prone, hardly exhaustive, and sometimes not recommended (e.g., decreasing readability, unnecessary waste of gas fees). As a result, many smart contracts do not perform manual safety checking exhaustively [7], [11]. Second, verification prevents bugs at compile time so that they can be fixed before deployment, but manual checking detects bugs only at runtime.

B. Analyzing Arithmetic Safety of Traditional Programs

Ensuring arithmetic safety has been studied extensively in the program analysis and verification communities [49], [50], [51], [52], [53], [54], [55], [56], [56], [57], [58]. Our work differs from them in two ways. First, we focus on smart contracts and provide a domain-specific algorithm. Second, to our knowledge, our CEGIS-style algorithm for verifying arithmetic safety is also new in this general context.

Astrée [49], [50] is a domain-specific static analyzer tailored to flight-control software. Sparrow [51] and Frama-C [52], [53] are domain-unaware static analyzers for C programs. Astrée, Sparrow, and Frama-C are based on abstract interpretation [59], [60]. Instead, we use a CEGIS-style algorithm because existing abstract domains such as intervals [59] and octagons [61] cannot capture domain-specific invariants (e.g., sum) of smart contracts. Furthermore, abstract interpretation cannot infer invariants that are useful in practice but not inductive with respect to their abstract semantics. While our approach is similar to the existing CEGIS approaches (e.g., [13], [14], [15]), to the best of our knowledge, its application to arithmetic safety verification has not been studied. Bounded verification approaches (e.g., [62], [63]) are different from our work as we perform unbounded verification. Our work is different from symbolic execution-based techniques [54], [55], [56], [56], [57], [58] or unsound static analysis [64], [65], as we aim to detect all bugs. A few techniques aim to fix integer overflow bugs [66], [67], [68], which may introduce unwanted changes in programs though useful.

VII. Conclusion

As smart contracts are safety-critical, formally verifying their correctness is of the greatest importance. In this paper, we presented a new and powerful verification algorithm for smart contracts. Its central feature is the ability to automatically infer hidden, in particular transaction, invariants of smart contracts and leverage them during the verification process. We implemented the algorithm in a tool, VeriSmart, for verifying arithmetic safety of Ethereum smart contracts and demonstrate its effectiveness on real-world smart contracts in comparison with existing safety analyzers. Our work shows a common yet significant shortcoming of existing approaches (i.e., inability to infer and use transaction invariants) and sheds light on the future development of automated tools for analyzing smart contracts.

Acknowledgment

We thank Junhee Lee and Minseok Jeon for their valuable comments on Proposition 1 and Appendix A. This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-01697, Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security and No.2019-0-00099, Formal Specification of Smart Contract).

References

[1] 2018, [Online; accessed 31-May-2019]. [Online]. Available: http://virtual-strategy.com/2018/12/05/hashcash-enters-malta-with-smart-contract-based-insurance-automation/

[2] Y. Hanada, L. Hsiao, and P. Levis, "Smart contracts for machine-to-machine communication: Possibilities and limitations," *CoRR*, vol. abs/1806.00555, 2018. [Online]. Available: http://arxiv.org/abs/1806.00555

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[4] 2016, [Online; accessed 31-May-2019]. [Online]. Available: https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

[5] 2018, [Online; accessed 31-May-2019]. [Online]. Available: https://blockexplorer.com/news/260-million-parity-proposes-eip-999-to-recover-frozen-multi-sig-funds/

[6] 2018, [Online; accessed 31-May-2019]. [Online]. Available: https://blog.peckshield.com/2018/04/25/proxyOverflow/

[7] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 664–676. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274737

[8] "Mythril classic: an open-source security analysis tool for ethereum smart contracts." 2018, [Online; accessed 31-May-2019]. [Online]. Available: https://github.com/ConsenSys/mythril-classic

[9] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[10] "Manticore: a symbolic execution tool for analysis of smart contracts and binaries," 2017, [Online; accessed 31-May-2019]. [Online]. Available: https://github.com/trailofbits/manticore

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

[12] L. Alt and C. Reitwiessner, "Smt-based verification of solidity smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 376–388.

[13] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 404–415, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168917.1168907

[14] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 287–296. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462174

[15] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, Berkeley, CA, USA, 2008, aAI3353225.

[16] 2018, [Online; accessed 31-May-2019]. [Online]. Available: https://github.com/VenusADLab/EtherTokens/blob/master/SHARKTECH/SHARKTECH.md

[17] "Solidity 0.5.3," 2019, [Online; accessed 31-May-2019]. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/index.html

[18] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 653–663. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274743

[19] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 67–82. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243780

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

[21] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *CoRR*, vol. abs/1809.03981, 2018.

[22] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Berlin, Heidelberg: Springer-Verlag, 2007.

[23] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[24] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, Jul. 2011, pp. 171–177, snowbird, Utah. [Online]. Available: http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf

[25] K. D. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 57–66. [Online]. Available: http://doi.acm.org/10.1145/53990.53996

[26] "Oyente: An analysis tool for smart contracts," 2018, [Online; accessed 31-May-2019]. [Online]. Available: https://github.com/melonproject/oyente

[27] "Etherscan," [Online; accessed 31-May-2019]. [Online]. Available: https://etherscan.io/

[28] "Zeus evaluation," 2018, [Online; accessed 31-May-2019]. [Online]. Available: https://goo.gl/kFNHy3

[29] [Online; accessed 31-May-2019]. [Online]. Available: https://github.com/ethereum/solidity/issues/6835

[30] [Online; accessed 31-May-2019]. [Online]. Available: https://blog.peckshield.com/2018/05/21/ceoAnyone/

[31] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 442–446.

[32] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "GASTAP: A gas analyzer for smart contracts," *CoRR*, vol. abs/1811.10403, 2018. [Online]. Available: http://arxiv.org/abs/1811.10403

[33] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 48:1–48:28, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3158136

[34] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 65–68.

[35] C. Reitwiessner, "Formal verification for solidity contracts." 2015, [Online; accessed 31-May-2019]. [Online]. Available: https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts

[36] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Financial Cryptography and Data Security*, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Cham: Springer International Publishing, 2017, pp. 520–535.

[37] 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, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: ACM, 2016, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[38] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds. Cham: Springer International Publishing, 2018, pp. 243–269.

[39] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: ACM, 2018, pp. 66–77. [Online]. Available: http://doi.acm.org/10.1145/3167084

[40] S. K. Lahiri, S. Chen, Y. Wang, and I. Dillig, "Formal specification and verification of smart contracts for azure blockchain," *CoRR*, vol. abs/1812.08829, 2018. [Online]. Available: http://arxiv.org/abs/1812.08829

[41] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *ASE*. ACM, 2018, pp. 259–269.

[42] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 343–361.

[43] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 243–262. [Online]. Available: http://doi.acm.org/10.1145/1640089.1640108

[44] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[45] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270. [Online]. Available: https://www.fstar-lang.org/papers/mumon/

[46] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 2018, pp. 204–217.

[47] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[48] "Openzeppelin: Safemath," 2018, [Online; accessed 31-May-2019]. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol

[49] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 196–207. [Online]. Available: http://doi.acm.org/10.1145/781131.781153

[50] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Why does astrée scale up?" *Formal Methods in System Design*, vol. 35, no. 3, pp. 229–264, 2009. [Online]. Available: https://doi.org/10.1007/s10703-009-0089-6

[51] "Sparrow," [Online; accessed 31-May-2019]. [Online]. Available: https://github.com/ropas/sparrow

[52] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015. [Online]. Available: https://doi.org/10.1007/s00165-014-0326-7

[53] "Frama-c: a source-code analyzer of c software." [Online; accessed 31-May-2019]. [Online]. Available: https://frama-c.com/index.html

[54] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with kint," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 163–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387897

[55] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society, 2009. [Online]. Available: https://www.ndss-symposium.org/ndss2009/intscope-automatically-detecting-integer-overflow-vulnerability-in-x86-binary-using-symbolic-execution/

[56] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 67–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855773

[57] Y. Moy, N. Bjørner, and D. Sielaff, "Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis," *Tech. Rep. MSR-TR-2009–57*, 2009.

[58] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 473–486. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694389

[59] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[60] ——, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79. New York, NY, USA: ACM, 1979, pp. 269–282. [Online]. Available: http://doi.acm.org/10.1145/567752.567778

[61] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006. [Online]. Available: https://doi.org/10.1007/s10990-006-8609-1

[62] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176.

[63] M. Clochard, J.-C. Filliâtre, and A. Paskevich, "How to avoid proving the absence of integer overflows," in *Verified Software: Theories, Tools,*

and Experiments, A. Gurfinkel and S. A. Seshia, Eds. Cham: Springer International Publishing, 2015, pp. 94–109.

[64] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy, "Flow-insensitive static analysis for detecting integer anomalies in programs," in *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*, ser. SE'07. Anaheim, CA, USA: ACTA Press, 2007, pp. 334–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1332044.1332098

[65] E. N. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop, "Using type qualifiers to analyze untrusted integers and detecting security flaws in c programs," in *Detection of Intrusions and Malware & Vulnerability Assessment*, R. Büschkes and P. Laskov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–16.

[66] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 439–452. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535888

[67] Z. Coker and M. Hafiz, "Program transformations to fix c integers," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 792–801. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486892

[68] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "Intpti: Automatic integer error repair with proper-type inference," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 996–1001. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155693

APPENDIX

A. *Preprocessing of Verification Conditions*

Given a basic path p, let F be a verification condition (either an inductiveness condition, i.e., $F = \text{GENVC}(p).1$, or a safety condition, i.e., $F \in \text{GENVC}(p).2$) that contains equalities of the form $\mathsf{sum}(x) = e$ for some mapping variable x and expression e. For simplicity, we assume that F does not contain primed instances (e.g., x', x'') of the mapping variable x. Let I be the set of variables in F used as indices of x. Then, we replace each equality $\mathsf{sum}(x) = e$ by G as follows. If $I = \emptyset$, we define G to be $G_1 \wedge G_2$ where $G_1 = (R_x = e)$, $G_2 = B_x$ (R_x and B_x are fresh variables, see Section III-D). If $I = \{i\}$ (i.e., I is a singletone set), we define G to be $G_1 \wedge G_2$ where $G_1 = (x[i] + R_x = e)$ and $G_2 = (x[i] + R_x \geq R_x \wedge B_x)$. Otherwise (i.e., $I = \{i_1, \ldots, i_n\}, n \geq 2$), we define G to be $G_1 \wedge G_2$ where

$$G_1 = \bigwedge_{\substack{a \in [1,m], \\ P_a = \\ \{\{i_1,\ldots\},\cdots,\{i_k,\ldots\}\}}} \Big(\big(\bigwedge_{I_u \in P_a} \big(\bigwedge_{i,j \in I_u} i = j \big) \wedge \bigwedge_{\substack{I_u, I_v \in P_a, \\ I_u \neq I_v}} \big(\bigwedge_{\substack{i \in I_u, \\ j \in I_v}} i \neq j \big) \big) \rightarrow x[i_1] + \cdots + x[i_k] + R_x = e \Big)$$

and

$$G_2 = \bigwedge_{\substack{a \in [1,m], \\ P_a = \\ \{\{i_1,\ldots\},\cdots,\{i_k,\ldots\}\}}} \Big(\big(\bigwedge_{I_u \in P_a} \big(\bigwedge_{i,j \in I_u} i = j \big) \wedge \bigwedge_{\substack{I_u, I_v \in P_a, \\ I_u \neq I_v}} \big(\bigwedge_{\substack{i \in I_u, \\ j \in I_v}} i \neq j \big) \big) \rightarrow H_{x,i,k} \wedge x[i_1] + \cdots + x[i_k] + R_x \geq R_x \Big) \Big) \wedge B_x.$$

$H_{x,i,k}$ is defined as *true* if $k = 1$, and defined as $\bigwedge_{c=2}^{k} x[i_1] + \cdots + x[i_c] \geq x[i_c]$ otherwise (i.e., $k \geq 2$). $P_1, \ldots, P_m$ are all possible partitions of the index variable set I, where a partition is a set of disjoint non-empty subsets of I such that the union of the subsets equals I. For example, given $I = \{i, j\}$, we have two partitions: $\{\{i, j\}\}$ and $\{\{i\}, \{j\}\}$. Also, given $I = \{i, j, k\}$, we have five partitions: $\{\{i, j, k\}\}$, $\{\{i\}, \{j, k\}\}$, $\{\{j\}, \{i, k\}\}$, $\{\{k\}, \{i, j\}\}$, and $\{\{i\}, \{j\}, \{k\}\}$.

Intuitively, G_1 asserts that the sum of distinct elements of x equals e, and G_2 asserts that overflows do not occur during computing the sum of the distinct elements. More specifically, using the partitions of I, we first consider all possible cases of (in)equalities among the variables in I; for each partition $P_a = \{I_1, \ldots, I_k\}$ (where $\uplus_{1 \leq i \leq k} I_i = I$), the variables in the same subsets have the same values (i.e., $\bigwedge_{I_u \in P_a} (\bigwedge_{i,j \in I_u} i = j)$), and the variables in different subsets have different values (i.e., $\bigwedge_{I_u, I_v \in P_a, I_u \neq I_v} (\bigwedge_{i \in I_u, j \in I_v} i \neq j)$). Then, for each partition case, we generate constraints on the distinct elements of x by selecting an index variable from each subset.

Example 4: Given a basic path p, suppose $F \in \text{GENVC}(p).2$ is given as follows:

$$\mathsf{sum}(y) = 100 \wedge y[i] \geq v \rightarrow y[j] + v \geq y[j]$$

In this case, the index variable set for y is $I = \{i, j\}$, because i and j are used as indices in $y[i]$ and $y[j]$, respectively. For I, we have two partitions $P_1 = \{\{i, j\}\}$ and $P_2 = \{\{i\}, \{j\}\}$, and thus we consider two cases: $i = j$ from P_1 and $i \neq j$ from P_2. Then, we replace $\mathsf{sum}(y) = 100$ by $G_1 \wedge G_2$ where G_1 is

$$(i \neq j \rightarrow y[i] + y[j] + R_y = 100) \wedge (i = j \rightarrow y[i] + R_y = 100)$$

and G_2 is

$$(i \neq j \rightarrow y[i] + y[j] \geq y[i] \wedge y[i] + y[j] + R_y \geq R_y) \wedge (i = j \rightarrow y[i] + R_y \geq R_y) \wedge B_y.$$

Finally, by replacing $\mathsf{sum}(y) = 100$ in F by $G_1 \wedge G_2$, we obtain the following F'

$$\begin{array}{l} \big((i \neq j \rightarrow y[i] + y[j] + R_y = 100) \wedge \\ (i = j \rightarrow y[i] + R_y = 100) \wedge \\ (i \neq j \rightarrow y[i] + y[j] \geq y[i] \wedge y[i] + y[j] + R_y \geq R_y) \wedge \\ (i = j \rightarrow y[i] + R_y \geq R_y) \wedge B_y \wedge \\ y[i] \geq v \big) \\ \rightarrow y[j] + v \geq y[j] \end{array}$$

which is satisfiable iff the original formula F is satisfiable.

B. *Proof of Proposition 1*

Proof by contradiction. Assume $p \implies q$:

$$\forall I. I \models \neg p \vee q. \tag{3}$$

From condition (ii) and (3), we have

$$I_p \models q \tag{4}$$

where I_p is an interpretation that makes the evaluation of p *true* (i.e., $I_p \models p$). From condition (ii), condition (iii), and (4), we have a x-variant of I_p, denoted as I'_p, such that

$$I'_p : I_p \lhd \{x \mapsto v\} \models \neg q \tag{5}$$

where $x \in \mathsf{FV}(q) \setminus \mathsf{FV}(p)$ and $v \in D_{I_p} \setminus \{\alpha_{I_p}[x]\}$. Since $I_p \models p$ and $x \notin \mathsf{FV}(p)$,

$$I'_p \models p. \tag{6}$$

Combining (5) and (6), we have $I'_p \models \neg(\neg p \vee q)$, which contradicts with the assumption (3). Thus $p \not\Rightarrow q$.

C. *More Examples of Validity Templates*

We provide three more examples that are important for performance. We assume that formula F below is in CNF (conjunctive normal form). We write $c \in F$ for indicating that F has clause c.

Example 5: Consider a template

$$\frac{\mathsf{sum}(x) = n \in F,\ x[p] \geq v \in F}{F \to x[q] + v \geq x[q]}\ n + n \geq n$$

where x is a mapping variable that maps address-typed index variables to 256-bit unsigned integer-typed variables, n is an integer constant (where $n + n$ does not overflow in unsigned 256-bit), and p and q are address-typed variables. The template above states that, when $\mathsf{sum}(x) = n$ and $x[p] \geq v$ hold in the precondition F, $x[q] + v \geq x[q]$ also holds for any index address-typed variable q. For example, we can use the rule to check that the VC

$$\cdots \wedge \mathsf{sum}(a) = 100 \wedge \cdots \wedge a[i] \geq k \wedge \cdots \to a[j] + k \geq a[j]$$

is valid without preprocessing the formula and invoking an SMT solver.

Example 6: Consider a template:

$$\frac{\mathsf{sum}(x) = y \in F,\ y = n \in F,\ x[p] \geq v \in F}{F \to x[q] + v \geq x[q]}\ n + n \geq n$$

where x is a mapping variable that maps address-typed index variables to 256-bit unsigned integer-typed variables, y and v are 256-bit unsigned integer-typed variables, n is an integer constant (where $n+n$ does not overflow in unsigned 256-bit), and p and q are address-typed variables. Note that the template above is similar to the one in Example 5, where $\mathsf{sum}(x) = n$ is changed into a combination of $\mathsf{sum}(x) = y$ and $y = n$. Using the template, we can prove the validity of the VC:

$$\cdots \wedge \mathsf{sum}(a) = b \wedge \cdots \wedge b = 100 \wedge \cdots \wedge a[i] \geq k \wedge \ldots \\ \to a[j] + k \geq a[j]$$

Example 7: Consider a template:

$$\frac{}{F \to n_1 + (x \% n_2) \geq n_1}\ n_1 + n_2 \geq n_1$$

where x is a 256-bit unsigned integer-typed variables, and n_1 and n_2 are integer constants (where $n_1 + n_2$ does not overflow in unsigned 256-bit). Using the validity template above, we can prove that $\cdots \to 48 + (y \% 10) \geq 48$ is valid.