Summary-Based Symbolic Evaluation for Smart Contracts

Yu Feng
yufeng@cs.ucsb.edu
University of California, Santa Barbara

Emina Torlak
emina@cs.washington.edu
University of Washington

Rastislav Bodik
bodik@cs.washington.edu
University of Washington

ABSTRACT

This paper presents Solar, a system for automatic synthesis of adversarial contracts that exploit vulnerabilities in a victim smart contract. To make the synthesis tractable, we introduce a query language as well as *summary-based symbolic evaluation*, which significantly reduces the number of instructions that our synthesizer needs to evaluate symbolically, without compromising the precision of the vulnerability query. We encoded common vulnerabilities of smart contracts and evaluated Solar on the entire data set from Etherscan. Our experiments demonstrate the benefits of summary-based symbolic evaluation and show that Solar outperforms state-of-the-art smart contracts analyzers, teether, Mythril, and ContractFuzzer, in terms of running time and precision.

ACM Reference Format:
Yu Feng, Emina Torlak, and Rastislav Bodik. 2020. Summary-Based Symbolic Evaluation for Smart Contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416646

1 INTRODUCTION

Smart contracts are programs running on top of blockchain platforms such as Bitcoin [19] and Ethereum [20]. They interact with each other to perform effective financial transactions in a distributed system without the intervention from trusted third parties (e.g., banks). A smart contract is written in a high-level programming language (e.g., Solidity [23]), and it is typically comprised of a unique address, persistent storage holding a certain amount of cryptocurrency (i.e., Ether in Ethereum), and a set of functions that manipulate the persistent storage to fulfill credible transactions without trusted parties. For contract-to-contract interaction, some functions are public and callable by other contracts. Thanks to the expressiveness afforded by the high-level programming languages and the security guarantees from the underlying consensus protocol, smart contracts have shown many attractive use cases, and their number has skyrocketed, with over 45 million [11] instances covering financial products, online gaming, real estate [15], shipping, and logistics [16].

Because all smart contracts deployed on a blockchain are freely accessible through their public methods, any functional bugs or vulnerabilities inside the contracts can lead to disastrous losses, as demonstrated by recent attacks [2, 4, 6, 27]. For instance, the code (simplified) in Figure 1 illustrates the notorious Reentrancy attack [6]. When the victim program (3) issues a money transaction to the attacker (2), it implicitly triggers the attacker's callback method, which invokes the victim's method (i.e., `withdraw`) again to make another transaction without updating the victim's balance. The attack maliciously extracted tokens from the victim and led to a financial loss of $150M in 2016. To make things worse, smart contracts are immutable—once they are deployed, fixing their bugs is extremely difficult due to the design of the consensus protocol.

Improving robustness of smart contracts is thus a pressing practical problem. Unsurprisingly, a complex vulnerability like Reentrancy typically involves interactions between multiple contracts, which requires an analyzer to model the inter-contracts communication and reason about the execution in a *precise* and *scalable* way. But existing tools either aggressively *overapproximate* the execution a smart contract and report warnings [34, 48] that do not correspond to feasible paths and therefore cannot be exploited, or they precisely enumerate [39, 42, 43] *concrete traces* of a smart contract, so cannot scale to large programs with many paths.

This paper presents Solar, a new point in the design space of smart contract analysis tools that achieves an effective trade-off among expressiveness, precision, and scalability. Solar provides the security analyst with a query language for expressing *vulnerability patterns* that can be exploited in an attack, as well as an automatic engine for *synthesizing* an attack program (if one exists) that exploits the given vulnerability. Our key insight is based on the observation that an attacker typically exploits the vulnerability by making a sequence of transitions (calls over public methods of the victim), in which storage states are preserved across different transitions. Because most types of vulnerabilities can be overapproximated through assertions over storage variables (Section 4.2), this insight motivates an effective summary-based symbolic evaluation technique where the summary of a method soundly models its side-effect over storage variables, which dramatically reduces the number of instructions that Solar has to re-evaluate symbolically. As a result, Solar is able to scale reasoning with better precision to large contracts that are out of reach of existing symbolic execution [42, 43] and fuzzing [39] tools. Furthermore, previous summarization techniques [26, 33] rely on symbolic execution and can therefore lead to summaries that are exponential in program size. Our technique relies on Rosette [47], a hybrid symbolic evaluator that combines symbolic execution and bounded model checking, to compute compact (i.e., polynomially-sized) and precise (i.e., encoding all feasible bounded paths) summaries at the procedure level. Using these summaries, Solar can perform precise all-paths analysis of a given contract while symbolically executing significantly fewer paths than Rosette alone.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3416646

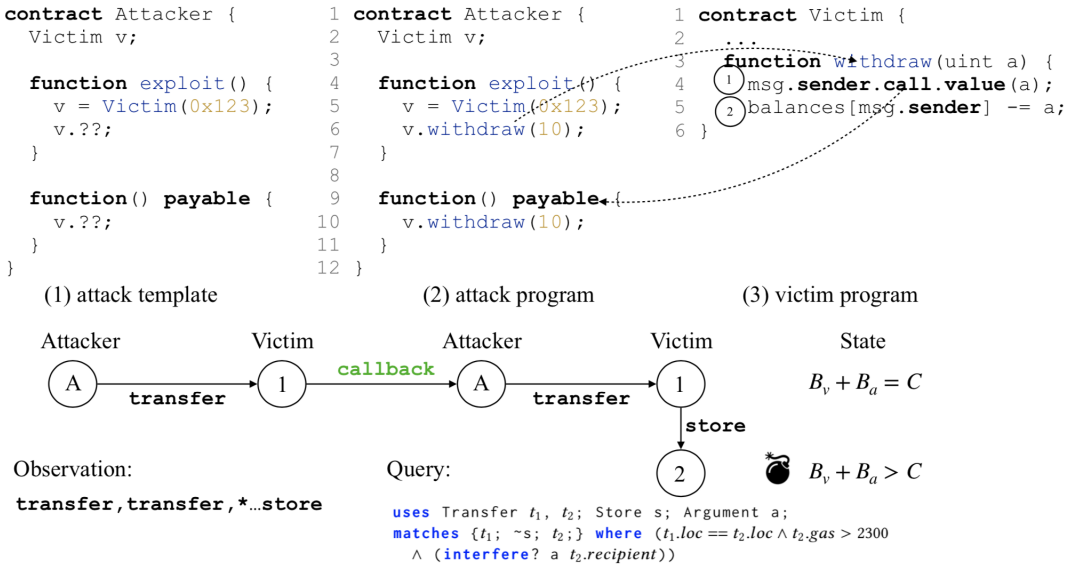

Figure 1: Sample contracts to show the Reentrancy attack.

To use our tool, a security analyst expresses a target vulnerability query (e.g., the reentrancy vulnerability) as a declarative specification. Solar then *synthesizes* an attack program that exploits the victim's public interface to satisfy the vulnerability query. Given this problem, a naive approach is to enumerate all possible candidate programs and then symbolically evaluate each of them to check if it satisfies the query. While precise, the naive approach fails to scale to realistic contracts.

Even with summarization, the search space is still too large for brute-force enumeration. To address this issue, we partition the search space by case splitting on the range of symbolic variables, which allows us to simultaneously explore multiple attack programs using Rosette's SMT-based symbolic evaluation engine [47].

We have evaluated Solar on the entire data set (>25K) from Etherscan [11], showing that our tool is expressive, efficient, and effective. Solar's query specification language is expressive in that it is rich enough to encode common vulnerabilities found in the literature (such as the Reentrancy attack [6], Time manipulation [17], and malicious access control [42]), Security Best Practices [10], as well as the recent BatchOverflow Bug [13] (CVE-2018–10299), which allows the attacker to create an arbitrary amount of cryptocurrency. Solar is efficient: on average it takes only 8 seconds to analyze a smart contract from Etherscan, which is four times faster than teether [42] and two orders of magnitude faster than ContractFuzzer [39]. Solar is also effective in that it significantly outperforms state-of-the-art smart contracts analyzers, namely, teether, Mythril, and ContractFuzzer, in terms of false positive and false negative rates. The approximate queries also enable Solar to generate compact summaries and explore deeper vulnerabilities in exchange for a minor loss in precision.

In summary, this paper makes the following contributions:

- We formalize the problem of exploit generation as a program synthesis problem and provide a query language for expressing common vulnerabilities in smart contracts as declarative specifications (Section 4.2).
- We propose a new summary-based symbolic evaluation technique for smart contracts that significantly reduces the number of paths that Solar has to execute symbolically (Section 5).
- We develop an efficient attack synthesizer based on the summary-based symbolic evaluation, which incorporates a novel combination of search space partitioning and parallel symbolic execution based on the semantics of candidate programs (Section 6.2).
- We perform a systematic evaluation of Solar on the entire data set from Etherscan. Our experiments demonstrate the substantial benefits of our technique and show that Solar outperforms three state-of-the-art smart contracts analyzers in terms of running time and precision. (Section 7).

2 BACKGROUND

We first review necessary background on smart contracts.

Smart Contract. Smart contracts are programs that are stored and executed on the blockchain. They are created through the transaction system on the blockchain and are immutable once deployed. Each smart contract is associated with a unique 160-bit address; a private persistent storage; a certain amount of cryptocurrency, expressed as a balance (i.e., Ether in Ethereum) held by the contract; and a piece of executable code that fulfills complex computations to manipulate the storage and balance. The code is typically written in a high-level Turing-complete programming language such as Serpent [22], Vyper [24], and Solidity [23], and then compiled to the

Ethereum Virtual Machine (EVM) bytecode [21], a low-level stack-based language. For instance, Figure 1 shows two smart contracts written in the Solidity programming language [23].

Application Binary Interface. In the Ethereum ecosystem, smart contracts communicate with each other using the Contract Application Binary Interface (ABI), which defines the signatures of public functions provided by the hosted contract. While ABI offers a flexible mechanism for communication, it also creates an attack surface for exploits that use the ABI of a given smart contract.

Threat Model. To synthesize an adversarial contract, we assume that the attacker can obtain the victim contract's bytecode and the ABI specifying its public methods. To confirm an adversarial contract is indeed an exploit, we must also be able to invoke public methods by submitting transactions over the Ethereum Blockchain. These requirements are easy to satisfy in practice.

3 OVERVIEW

In this section, we give an overview of our approach with the aid of a motivating example.

3.1 Smart Contract Vulnerabilities

A security analyst, Alice, can specify various types of vulnerabilities that may appear in a smart contract. For instance, Figure 1 shows a simplified example of a Reentrancy attack. The `withdraw` function does two steps: ① send a given amount of Ether to the caller, and ② update the storage state to reflect the new balance. At any point, the total amount of balances of the victim and attacker should remain the same (i.e., $B_v + B_a = C$). However, since ① happens before updating the state in ②, an attacker can re-enter the `withdraw` function again through the anonymous callback function triggered by ①. As a result, the execution of the attack program can lead to an inconsistent state (i.e., $B'_v + B'_a > C$), which enables the attacker to extract a large amount of Ether from the victim.[1]

To automatically generate exploits for the Reentrancy vulnerability, Alice first specifies a *query* that *characterizes* the semantics of Reentrancy. As shown in the lower part of Figure 1, the attack can be summarized using a sequence of key statements between the victim and the attacker, i.e., two or more `transfer` [2] instructions followed by a `store` operation, which can be expressed using the first-order formula [3] in Figure 1.

Once Alice expresses the Reentrancy vulnerability, the next step is to construct an attack to confirm that the vulnerability indeed exists in the victim contract. Alice can leverage existing symbolic execution tools [12, 42, 43] to generate exploits for simple properties such as attack-control [42]) in a *single contract*. But for complex vulnerabilities that require reasoning about interactions among multiple contracts (e.g., attacker versus victim in Reentrancy or caller versus callee in Parity Multisig [14]), existing tools provide either no support [42] or very limited support that leads to high rates [43] of false positives and negatives (as shown in Section 7.1). Yet Alice can easily initialize the boilerplate code for basic interactions, like the "attack template" on the left hand side of Figure 1.

[1] Ethereum's gas mechanism ensures that this callback loop terminates.
[2] We use `transfer` to denote the `call` instruction in EVM.
[3] Solar converts a query into its corresponding FOL formulas through a syntax-directed translation.

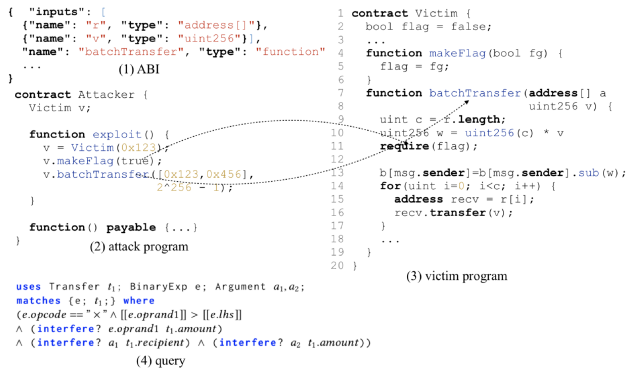

Figure 2: An example to show the BatchOverflow attack.

What she needs is an efficient way to fill in the details of the attack program, which involves exploring the space of all programs that can be obtained by completing the template with the methods from the victim's interface.

3.2 Solar

Solar helps automate this process by searching for attacks that exploit a given vulnerability in a victim contract. The tool takes as input a potential vulnerability $\mathcal{V}$ expressed as a declarative specification. If $\mathcal{V}$ exists in the victim contract, Solar automatically synthesizes an *attack program* that exploits $\mathcal{V}$. An attacker interacts with a vulnerable contract through its public methods defined in the ABI. Therefore, our goal is to construct an attack program that exploits the victim's ABI and that contains at least one concrete trace where $\mathcal{V}$ holds.

To achieve this goal, Solar models the executions of a smart contract as *state transitions* over registers, memory, and storage. The vulnerability $\mathcal{V}$ is expressed in Racket [5] as a boolean predicate over these state transitions. The technical challenge addressed by Solar is to efficiently search for an attack program where $\mathcal{V}$ holds.

To illustrate the difficulty of this task, consider the problem of synthesizing an attack program that exploits the BatchOverflow vulnerability (CVE-2018–10299) [13] in Figure 2. The attack program performs a complex three-step interaction with the victim contract. First, the attacker must set the storage variable `flag` to `true` to pass the check at line 11. Next, it needs to assign a large number to `v` that leads to an overflow at line 10. Finally, it specifies the attacker's address as the beneficiary of the transaction (line 16). Synthesizing this attack program involves discovering which methods to call, in what order, and with what arguments.

The naive approach to solving this problem is to generate all possible *concrete programs* and explore the space of their *concrete traces*. This approach suffers from two sources of exponential explosion. First, there are $O(n^k)$ concrete programs of length k for a victim contract with n public methods. Second, the number of concrete traces in each of these programs is exponential in the size of the program's global control-flow graph obtained by inlining all method calls.

To address the trace explosion challenge, Solar employs a novel summary-based symbolic evaluation technique presented in Section 5. Intuitively, this technique enables Solar to preserve only

$\langle var \rangle$ ::= `def-sym` id τ where $\tau \in$ {**boolean**, **number**}
$\langle pc \rangle$ ::= $\langle const \rangle \mid \langle var \rangle$
$\langle expr \rangle$ ::= $\langle const \rangle \mid \langle var \rangle \mid \langle expr \rangle \oplus \langle expr \rangle$
($\oplus \in \{+, -, \times, /, \vee, \wedge, \ldots\}$)
$\langle stmt \rangle$ ::= $\langle var \rangle := \langle expr \rangle$
| $\langle var \rangle :=$ **mload** $\langle var \rangle$ | **mstore** $\langle var \rangle \langle var \rangle$
| $\langle var \rangle :=$ **sload** $\langle var \rangle$ | **sstore** $\langle var \rangle \langle var \rangle$
| $\langle var \rangle :=$ {**balance**, **gas**, **address** }
$\langle stmts \rangle$ ::= $\langle stmt \rangle \mid \langle stmt \rangle; \langle stmts \rangle$ | **sha3** $\langle var \rangle \langle var \rangle$
| **jumpI** $\langle pc \rangle \langle expr \rangle$ | **jump** $\langle pc \rangle$ | **no-op**
| **transfer** $\langle var \rangle \langle var \rangle \langle \ldots \rangle$ | **selfdestruct** $\langle var \rangle$
$\langle param \rangle$ ::= $\langle var \rangle$
$\langle params \rangle$ ::= $\langle param \rangle \mid \langle param \rangle, \langle params \rangle$
$\langle prog \rangle$ ::= $\lambda \langle params \rangle. \langle stmts \rangle$

Figure 3: Intermediate language for smart contract

those state transitions that are persistent across different transactions and are *sufficient* to answer the vulnerability query.

To address the program explosion challenge, Section 6 introduces two additional optimizations. First, instead of exploring the space of concrete programs, we leverage Rosette [47] to partition this space into a small set of *symbolic programs* (Section 6.1). Second, instead of executing each symbolic program *sequentially*, we partition the search space by case splitting on the range of symbolic variables, which enables Solar to simultaneously explore multiple symbolic candidates (Section 6.2).

(a) Solidity program

```
1  require(_amount > 0);
2  vesting.amount = _amount.sub(1);
3  transfer(msg.sender,_to,vesting.amount);
4  uint256 v1 = _amount - 15;
5  uint256 wei = v1;
6  uint t1 = vesting.startTime;
7  emit VestTransfer(msg.sender, _to, wei, t1, _);
```

(b) Symbolic evaluation

```
1  assert(_amount > 0);
2  r1 := _amount - 1;
3  sstore(vesting.amount, _amount - 1);
4  transfer(msg.sender, _to, _amount - 1);
5  r2 := amount - 15;
6  r3 := amount - 15;
7  r4 := sload(vesting.startTime);
8  no-op;
```

(c) Summary extraction

```
1  sstore(vesting.amount, Γ_S[_amount] − 1)@(Γ_S[_amount] > 0);
2  transfer(Γ_S[msg.sender], Γ_S[_to], Γ_S[_amount] − 1)@(Γ_S[_amount] > 0)
       ;
```

(d) Summary interpretation

```
1  if (Γ[_amount] > 0) sstore(vesting.amount, Γ[_amount] − 1);
2  if (Γ[_amount] > 0) transfer(Γ[msg.sender], Γ[_to], Γ[_amount] − 1);
```

Figure 4: From Standard to Summary-Based Symbolic Evaluation

4 PROBLEM FORMULATION

This section formalizes the semantics of smart contracts, shows how to express smart contract vulnerabilities in Solar, and defines the problem of synthesizing an attack contract that exploits a given vulnerability.

4.1 Smart Contract Language

Figure 3 shows the core features of our intermediate language for smart contracts. This language is a superset of the EVM language. It includes standard EVM bytecode instructions such as assignment (`x := e`), memory operations (`mstore,mload`), storage operations (`sstore,sload`), hash operation (`sha3`), sequential composition ($s_1; s_2$), conditional (`jumpi`) and unconditional jump (`jump`). It also includes the EVM instructions specific to smart contracts: `transfer` denotes all functions that send tokens between different addresses, `balance` accesses the current account balance, and `selfdestruct` terminates a contract and transfers its balance to a given address. Finally, our language extends EVM with features that facilitate symbolic evaluation, including *symbolic variables* (introduced by `def-sym`) and *symbolic expressions* (obtained by operating on symbolic variables) whose concrete values will be determined by an off-the-shelf SMT solver [44].

We define the operational semantics of each statement in Figure 3 based on the standard defined by the EVM yellow paper [7]. The semantics is lifted to work on symbolic values in the standard way [47]. The meaning of a statement is given by a *state transition* rule that specifies the statement's effect on the *program state*. We define states and transitions as follows.

Definition 4.1. (**Program State**) The *Program State* Γ consists of a stack E, memory M, persistent storage S, global properties (e.g., balance, address, timestamp) of a smart contract, and the program counter pc. We use e_i, m_i, and μ_i to denote variables from the stack, memory, and storage, respectively.

A program state also includes a model of the gas system in EVM, but we omit this part of the semantics to simplify the presentation. If a state maps a variable to a symbolic expression, we call it a *symbolic state*.

Definition 4.2. (**State transition over statement** s) A *State Transition* $\mathcal{T}$ over a statement s is denoted by a judgment of the form $\Gamma \vdash s : \Gamma', v$. The meaning of this judgment is the following: assuming we successfully execute s under program state Γ, it will result in value v and the new state is Γ'.

Example 4.3. Figure 4a shows a smart contract written in Solidity. To analyze this contract, Solar first translates it to the program in Figure 4b, using the intermediate language in Figure 3. The resulting program is then evaluated symbolically in an environment Γ that binds `_amount` to a fresh symbolic number. For instance, after executing line 2 in Figure 4b, register `r1` holds a symbolic value represented by Γ[`_amount`] − 1. Since Solar does not model the event system in Solidity, we turn the corresponding instructions (e.g., line 7 in Figure 4b) into `no-ops`.

Definition 4.4. (**Abstract execution trace**) An abstract execution trace $\mathcal{R}$ contains a list of events (i.e., statements) that are of interest. Each event has an event type representing the type of statement, and a list of attributes.

4.2 Smart Contract Vulnerabilities

We now describe how to express smart contract vulnerabilities in Solar and what it means for a vulnerability to appear in a program.

Figure 5 shows our query language over program traces. A query consists of three parts. The **uses** block declares typed variables, which are matched against variables or statements appearing in the program. The **matches** block specifies a sequence of statements that are matched against the program trace. The **where** clause further refines the search criteria by imposing constraints over the matched statements.

Query variables. Query variables in the **uses** block correspond to variables or statements in the program trace. Common variables include statements, storage variables, arguments, etc.

Statements. Statements in the query language correspond to events in the execution trace discussed in Section 4. In particular, an event is of type record whose fields are properties of that event. Table 1 lists the fields of some representative statements appearing in the query. Furthermore, a `seqStmt` such as `a;b` specifies that the event `a` happens before `b`. Finally, the exclusion operator "~" is used to prohibit an event from appearing in the trace.

Conditional clauses. The criteria of a query can be further refined using the *conditional clauses* in the **where** block. In particular, a conditional clause is a boolean expression whose sub-expressions are constants, query variables, fields of query variables, or custom predicate like `interfere` which we introduce later.

$\langle query \rangle$ ::= $\langle$*uses declList;*$\rangle$
| $\langle$*matches {seqStmt}*$\rangle$
| $\langle$*where cond*$\rangle$

$\langle declList \rangle$ ::= $\langle$*typeName id (,id)**$\rangle$
$\langle typeName \rangle$ ::= $\langle id \rangle$
$\langle stmt \rangle$ ::= $\langle transfer \rangle$ | $\langle sstore \rangle$ | $\langle jump \rangle$ | $\langle binaryExp \rangle$ | $\langle$*~stmt*$\rangle$ ...
$\langle seqStmt \rangle$ ::= $\langle stmt \rangle$ | $\langle$*stmt;stmt*$\rangle$
$\langle cond \rangle$ ::= $\langle E \rangle \oplus \langle E \rangle$ ($\oplus \in \{+, -, >, \neq, \vee, \wedge, ...\}$)
$\langle E \rangle$ ::= $\langle const \rangle$ | [[var]] | $\langle var \rangle$
| $\langle$*fieldAccess*$\rangle$ | (interfere? $\langle E \rangle$ $\langle E \rangle$)
$\langle var \rangle$ ::= $\langle local \rangle$ | $\langle argument \rangle$
$\langle$*fieldAccess*$\rangle$ ::= $\langle$*id.id*$\rangle$
$\langle id \rangle$ ::= $\langle$*A-Za-z*$\rangle^*$

Figure 5: Query language for Solar

Compilation of query. Solar converts query into corresponding FOL formulas through a syntax-directed translation. For queries that contain quantifiers, we use skolemization to make them quantifier-free (or reject them if they cannot be skolemized).

The rest of this section introduces a few representative vulnerabilities, and shows how they are encoded as formulas in Solar. But first, we introduce an auxiliary function `interfere?` which will be used by several vulnerabilities.

Definition 4.5. **(Interference)** A symbolic variable v interferes with a symbolic expression e if they satisfy the following constraint: $\exists v_0, v_1.\ e[v_0/v] \neq e[v_1/v] \wedge (v_0 \neq v_1)$

Fields of transfer statement	
`sender`	sender's address
`recipient`	target's address
`loc`	program counter of the statement
`gas`	gas budget for the transfer
`amount`	amount of tokens
`ret`	return value of the statement
Fields of jump statement	
`condVar`	condition variable of jump statement
`target`	target address
Fields of sstore statement	
`name`	name of storage variable
`value`	new value that is used
Fields of binary statement	
`lhs`	variable that is assigned
`opcode`	opcode of the binary statement
`oprand1`	the first operand
`oprand2`	the second operand

Table 1: Fields of core statements appearing in the query language

Intuitively, changing v's value will also affect e's output, which is denoted as "(interfere? v e)". Interference precisely captures the data- and control-dependencies between two expressions and turns out to be the *necessary condition* of many exploits.

Section 3 describes the BatchOverflow vulnerability, which enables an attacker to perform a multiplication that overflows and transfers a large amount of tokens on the attacker's behalf. This vulnerability can be formalized as follows:

Vulnerability 1. BatchOverflow

```
uses Transfer t1; BinaryExp e; Argument a1, a2;
matches {e; t1;} where
(e.opcode == "×" ∧ [[e.oprand1]] > [[e.lhs]]
∧ (interfere? e.oprand1 t1.amount)
∧ (interfere? a1 t1.recipient) ∧ (interfere? a2 t1.amount))
```

The query specifies that the victim program contains a `transfer` instruction whose beneficiary and value can be controlled by the attacker. Furthermore, the transaction value is also influenced by a variable from an arithmetic operation that overflows.

An *Unchecked-send Vulnerability* occurs when the programmer fails to check the return values of critical instructions such as `delegatecall` and `call`. If these instructions result in runtime errors, the programmer is responsible for manually checking their return values and restoring the program state. Failing to do so can lead to unexpected behavior [18]. We formalize the absence of this check as follows:

Vulnerability 2. **Unchecked-send (Gasless-send)**

```
uses Transfer t; Jump j;
matches { t; ~j;} where ((interfere? t.ret j.condVar))
```

Here, the return value of a `transfer` instruction does not *interfere with* the conditional variables of any *conditional jump* statements. In other words, this return value is not checked.

The Reentrancy vulnerability (introduced in Section 1) occurs when an attacker's call is allowed to repeatedly make new calls to the same victim contract without updating the victim's balance. It can be overapproximated as follows:

Vulnerability 3. **Reentrancy**

```
uses Transfer t1, t2; Store s; Argument a;
matches {t1; ~s; t2;} where (t1.loc == t2.loc ∧ t2.gas > 2300
  ∧ (interfere? a t2.recipient))
```

In other words, let trace $\mathcal{R}$ contains a sequence instructions that include multiple `transfer` statements that share the same program counter, if there is no `store` statement between the two `transfer` functions that has the minimum gas (i.e., 2300), then there may exist a Reentrancy vulnerability.

4.3 Attack Synthesis

Given a vulnerability query, we are interested in synthesizing an attack program that can exploit this vulnerability in a victim contract. The basic building blocks of an attack program are called *components*, and each component C corresponds to a public method provided by the victim contract. We use Υ to denote the union of all publicly available methods.

Definition 4.6. **(Component)** A *Component* C from an ABI configuration is a pair (f, τ) where: 1) f is C's name, and 2) τ is the type signature of C.

Example 4.7. Consider the ABI configuration in Figure 2. Its first element declares a component for the problematic `batchTransfer` method. This component takes inputs as an array of `address` and a 256-bit integer (`uint256`).

We represent a set of candidate attack programs as a *symbolic program*, which is a sequence of *holes* to be filled with components from Υ. The synthesizer fills these holes to obtain a *concrete program* that exploits a given vulnerability.

Definition 4.8. **(Symbolic Attack Program)** Given a set of components $\Upsilon = \{(f_1, \tau_1), \ldots, (f_N, \tau_N)\}$, a *symbolic attack program* $\mathcal{S}$ for Υ is a sequence of *statement holes* of the form

$$\mathsf{choose}(f_1(\vec{v}_{\tau_1}), \ldots, f_N(\vec{v}_{\tau_N}));$$

where $f_i(\vec{v}_{\tau_i})$ stands for the application of the i-th component to fresh symbolic values of types specified by τ_i.

Definition 4.9. **(Concrete Attack Program)** A *concrete attack program* for a symbolic program $\mathcal{S}$ replaces each hole in $\mathcal{S}$ with one of the specified function calls, and each symbolic argument to a function call is replaced with a concrete value.

Example 4.10. Here is a symbolic program that captures the attack candidate in Fig 2:

```
choose(makeFlag(x1), batchTransfer(y1,z1));
choose(makeFlag(x2), batchTransfer(y2,z2));
```

And here is a concrete attack program for this symbolic attack:

```
makeFlag(true);
batchTransfer([0x123,0x345], 2^256 − 1);
```

```
1  (define (get-summary s φ)
2    (match s
3      [transfer(x, y, z)  transfer(Γ_S(x), Γ_S[y], Γ_S[z])@φ]
4      [sstore(x, y)       sstore(x, Γ_S[y])@φ]
5      [_                  #f]))
```

Figure 6: Procedure for summary generation.

The choose construct is a notational shorthand for a conditional statement that guards the specified choices with fresh symbolic booleans. For example, $\mathsf{choose}(e_1, e_2)$ stands for the statement `if` b_1 `then` e_1 `else` e_2, where b_1 is a fresh symbolic boolean value. A concrete attack program therefore substitutes concrete values for the implicit choose guards and the explicit function arguments of a symbolic attack program.

The goal of attack synthesis is to find a concrete program P for a given symbolic program $\mathcal{S}$ such that P reaches a state satisfying a desired vulnerability query.

Definition 4.11. **(Problem Specification)** The specification for our *attack synthesis* problem is a tuple $(\Gamma_0, \mathcal{V}, \mathcal{S})$ where:

- $\mathcal{S}$ is a symbolic attack program for the set of components Υ of a victim contract V.
- Γ_0 is the initial state of the symbolic attack program, obtained by executing the victim's initialization code.
- $\mathcal{V}$ is a first-order formula over the (symbolic) program state $[[\mathcal{S}]]_\Gamma$ reachable from Γ_0 by the attack program $\mathcal{S}$.

Definition 4.12. **(Attack Synthesis)** Given a specification $(\Gamma_0, \mathcal{V}, \mathcal{S})$, the *Attack Synthesis problem* is to find a *concrete attack program* P for $\mathcal{S}$ such that: 1) $[[P]]_{\Gamma_0} = \Gamma$, and 2) $\Gamma \models \mathcal{V}$. In other words, executing P from the initial state Γ_0 results in a program state Γ that satisfies $\mathcal{V}$.

5 SUMMARY-BASED SYMBOLIC EVALUATION

Solving the attack synthesis problem involves searching for a concrete program P in the space of candidate attacks defined by a symbolic program $\mathcal{S}$. Solar delegates this search to an off-the-shelf SMT solver, by using symbolic evaluation to reduce the attack synthesis problem to a satisfiability query. Given a specification $(\Gamma_0, \mathcal{V}, \mathcal{S})$, Solar evaluates $\mathcal{S}$ on the state Γ_0 to obtain the state $[[\mathcal{S}]]_{\Gamma_0}$, and then uses the solver to check the satisfiability of the formula $\exists \vec{v}.\mathcal{V}([[\mathcal{S}]]_{\Gamma_0})$, where $\vec{v}$ denotes the symbolic variables in $\mathcal{S}$. A model of this formula, if it exists, binds every variable in $\vec{v}$ to a concrete value, and so represents a concrete attack program P for $\mathcal{S}$ that triggers the vulnerability $\mathcal{V}$. But computing $[[\mathcal{S}]]_{\Gamma_0}$ is expensive as it relies on symbolic evaluation [47]. In particular, evaluating a choose statement in $\mathcal{S}$ involves symbolically evaluating each function call in that statement. So, for a symbolic program of length K, every public function in the victim contract must be symbolically executed K times on different symbolic arguments. As we will see in section 7, this direct approach to evaluating $\mathcal{S}$ does not scale to real contracts that contain a large number of complex public functions. To mitigate this issue, we use a summary-based symbolic evaluation that performs symbolic execution of each public method only once.

Our approach is based on the following insight. An attack program performs a sequence of transactions—i.e., method invocations—that manipulate the victim's persistent storage and global properties. The transactions that comprise an attack exchange data and influence each other's control flow exclusively through these two parts of the program state. So, if we can faithfully summarize the effects of a public method on the persistent storage and global properties, evaluating this summary on the symbolic arguments passed to the method is equivalent to symbolically executing the method itself.

Definition 5.1. A summary $\mathcal{M}$ in our system is a pair $s@\phi$ where s represents a statement that has a side effect on the persistent state (i.e., storage and global properties) of a smart contract, and ϕ denotes the path condition under which s is executed.

We generate such faithful method summaries in two steps. First, we evaluate the method on a program state Γ_S that maps every state variable (i.e., persistent storage location, global property, etc.) to a fresh symbolic variable of the right type. This step produces a path condition and symbolic inputs for each instruction that capture every possible way to reach and execute the instruction within the given method. Next, we use the procedure in Figure 6 to generate the method summary.[4] Given a storage-store instruction `sstore(x,y)` and its path condition, we generate a "summary sstore" statement (i.e., $\overline{sstore}$) that takes as input the name of the storage variable (i.e., x) and the symbolic expression $\Gamma_S[y]$ held in the register y. Similarly, given a `call(gas,addr,value)` instruction and path condition, we emit its "summary call" statement (i.e., $\overline{call}$) that takes as input the symbolic expressions of the instruction's gas consumption, recipient address, and amount of cryptocurrency, respectively. All other instructions are omitted from the summary since they have no effect on the persistent state. By construction, our summary therefore precisely captures all of the method's effects on the persistent state, and the summaries are polynomially-sized as guaranteed by Rosette's symbolic evaluator [47].

Example 5.2. Recall that we introduce the following code snippet in Figure 4b:

```
1 assert(_amount > 0);
2 r1 := _amount - 1;
3 sstore(vesting.amount, _amount - 1);
4 transfer(msg.sender, _to, _amount - 1);
5 r2 := amount - 15;
6 r3 := amount - 15;
7 r4 := sload(vesting.startTime);
8 no-op;
```

Then using the rule in Figure 6, Solar generates the following summary:

$\overline{sstore}$(vesting.amount, Γ_S[_amount] − 1)@(Γ_S[_amount] > 0);
$\overline{transfer}$(Γ_S[msg.sender], Γ_S[_to], Γ_S[_amount] − 1)@(Γ_S[_amount] > 0);

In particular, our tool summarizes the side effects of the `transfer` and `sstore` instructions at lines 2 and 3 in Figure 4b, respectively. The remaining instructions (e.g., statements from line 5 to 8) are omitted from the summary because they have no persistent side effects.

Once Solar generates the summary for each procedure, we still need to adjust the symbolic evaluation engine to take advantage of the summaries. Given a method summary and a program state Γ, we use the procedure in Figure 7 to reproduce the effects of executing the method symbolically on Γ as follows. Recall that we generate the summary by executing the method on a fully symbolic state $\Gamma_S = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, so every path condition and symbolic expression in the summary is given in terms of the symbolic variables $v_1, \ldots, v_n$. Our summary interpretation procedure works by substituting each v_i in an instruction's path condition and inputs with its corresponding value in Γ, i.e., $\Gamma[x_i]$. The resulting instruction summary $s_\Gamma@\phi_\Gamma$ is therefore expressed in terms of Γ, so applying its side effects s_Γ under the path condition ϕ_Γ is equivalent to executing the instruction s in the original method on the state Γ. Since we interpret every instruction in the summary in this way, the combined effect on the persistent state is equivalent to executing the original method symbolically on Γ.

```
1 (define (interpret-summary s@φ Γ)
2   (define sΓ@φΓ (substitute s@φ Γ))
3   (match sΓ
4     [transfer(xΓ, yΓ, zΓ) (when φΓ transfer(xΓ, yΓ, zΓ))]
5     [sstore(x, yΓ) (when φΓ sstore(x, yΓ))]
6     [_  no-op]))
```

Figure 7: Procedure for summary interpretation

```
1 (define (solar 𝒱 ϒ K)
2  (define program (for/list ([i K]) (apply choose* ϒ)))
3  (define i-pstate (get-initial-state ϒ))
4  (define o-pstate (interpret program i-state))
5  (define binding (solve (assert (𝒱 o-pstate))))
6  (evaluate program binding))
```

Figure 8: Solar implementation in Rosette.

Example 5.3. Figure 4d shows an example for interpreting the summary in Figure 4c by applying the procedure in Figure 7. Specifically, given an environment Γ and the `transfer` summary at line 2 in Figure 4c, we first generate an `if` statement guarded by the path condition ϕ in Γ, then in the body of the `if` statement, we symbolically evaluate the `transfer` statement in the environment Γ.

6 IMPLEMENTATION

This section discusses the design and implementation of Solar, as well as two key optimizations that enable our tool to efficiently solve the synthesis attack problem.

6.1 Symbolic Computation Using Rosette

Solar leverages Rosette [47] to symbolically search for attack programs. Rosette is a programming language that provides facilities for symbolic evaluation. Rosette programs use assertions and symbolic values to formulate queries about program behavior, which are then solved with off-the-shelf SMT solvers. For example, the `(solve expr)` query searches for a binding of symbolic variables to concrete values that satisfies the assertions encountered during the symbolic evaluation of the program expression `expr`. Solar uses the `solve` query to search for a concrete attack program.

Figure 8 shows the implementation of Solar in Rosette. The tool takes as input a vulnerability specification $\mathcal{V}$, the components Υ of a

[4] We omit the details of other side-effecting instructions for simplicity.

victim program, and a bound K on the length of the attack program. Given these inputs, line 2 uses Υ to construct a symbolic attack `program` of length K. Next, lines 3 runs the victim's initialization code to obtain the initial program state, `i-pstate`, for the attack. Then, line 4 evaluates the symbolic attack `program` on the initial state to obtain a symbolic output state, `o-pstate`. Finally, lines 5-6 use the `solve` query to search for a concrete attack program that satisfies the vulnerability assertion.

The core of our tool is the *interpreter* for our smart contract language (Figure 3), which implements the semantics from the EVM yellow paper [7]. We use this interpreter to compute the symbolic summaries of the victim's public methods (Section 5) and to evaluate symbolic attack programs. The interpreter itself does not implement symbolic execution; instead, it uses Rosette's symbolic evaluation engine to execute programs in our language on symbolic values.

Another key component of Solar is the *translator* that converts EVM bytecode into our language (Figure 3). The translator leverages the Vandal Decompiler [34] to soundly convert the stack-based EVM bytecode into its corresponding three-address format in our language. The jump targets are resolved through abstract interpretation [32]. We use the translator to convert victim contracts to the Solar language for attack synthesis. Both the translator and the interpreter support all the instructions defined in the Ethereum specification [21].

6.2 Parallel Synthesis using Hoisting

Solar uses summary-based symbolic evaluation to efficiently reduce attack synthesis problems to satisfiability queries. But the resulting queries can still be too difficult to solve in practice, especially when the victim contract has many public methods. To further improve performance, Solar exploits the structure of symbolic attack programs (Definition 4.8) to decompose the single `solve` query in Figure 8 into multiple smaller queries that can be solved quickly and in parallel, without missing any concrete attacks.

The basic idea is as follows. Given a set of N components and a bound K on the length of the attack, line 2 creates a symbolic attack program of the following form:

$$\texttt{choose}_1(f_1(\vec{v}_{1\tau_1}), \ldots, f_N(\vec{v}_{1\tau_N}));$$
$$\vdots$$
$$\texttt{choose}_K(f_1(\vec{v}_{K\tau_1}), \ldots, f_N(\vec{v}_{K\tau_N}));$$

This symbolic attack encodes a set of concrete attacks that can also be expressed using N^K symbolic programs that fix the choice of the method to call at each line, but leave the arguments symbolic. So, we can enumerate these N^K programs and solve the vulnerability query for each of them, instead of solving the single query at line 5. This approach essentially *hoists* the symbolic boolean guards out of the `choose` statements in the original query, and Solar explores all possible values for these guards explicitly, rather than via SMT solving.[5] As we show in Section 7, hoisting the guards leads to significantly faster synthesis, both because it enables parallel solving of the smaller queries, and because the smaller queries can be solved quickly.

[5]For practical efficiency, our implementation hoists the guards to generate N^K/c symbolic programs, where c is the number of available cores.

6.3 Practical EVM fragment

In this section, we briefly illustrate how Solar handles other challenging features of EVM.

Loops. Similarly to other analyzers based on symbolic execution, Solar unrolls all potentially unbounded loops K times. We use $K = 2$ as the default bound for unrolling.

SHA and Storage access. In the EVM bytecode, the address of an array or map element is determined by the following function:

$$a[i] := \textit{SHA-256(id(a))} + n \times i$$

Here, *SHA-256(id(a))* stands for the SHA-256 hash of the array's identifier, n is the size of the elements stored in the array, and i is the array index. Reasoning about this function directly is intractable for solvers. Solar circumvents this problem by leveraging uninterpreted functions to soundly model both the SHA-256 hash and the address computation function. That is, two addresses are the same if they share the same array identifier, index, and element size.

Gas consumption. Solar's program state tacks gas usage by accumulating the cost of instructions during symbolic evaluation. If a transaction runs out of gas in the middle of the evaluation, Solar terminates it with an "out of gas" assertion failure.

7 EVALUATION

We evaluated Solar by conducting a set of experiments that are designed to answer the following questions:

- RQ1: *Effectiveness*: How does Solar compare against state-of-the-art analyzers for smart contracts?
- RQ2: *Efficiency*: How much does summary-based symbolic evaluation improve the performance of Solar?

To answer these questions, we perform a systematic evaluation by running Solar on the entire set of smart contracts from Etherscan [11]. Using a snapshot from Feb 13 2019, we obtained a total of 25,983 smart contracts (duplicate contracts were removed) with publicly available source code. Solar starts from attack programs of size one and gradually increases the size until finding the exploit or running out of time. All experiments in this section are conducted on a `t3.2xlarge` machine on Amazon EC2 with an Intel Xeon Platinum 8000 CPU and 32G of memory, running the Ubuntu 18.04 operating system and using a timeout of 10 minutes for each smart contract.

7.1 Comparison with Existing Tools

To show the advantages of our proposed approach, we compare Solar against three state-of-the-art analyzers for exploits generation: Mythril and teether, based on symbolic execution, and ContractFuzzer, based on dynamic random testing.

Comparison with Mythril. We first compare with Mythril [12] [6] by generating exploits for the reentrancy vulnerability. Mythril takes as input a smart contract and checks whether there are concrete traces that match the tool's predefined security properties. If so, the tool returns a counterexample as the exploit. We evaluate

[6]Since both Solar and Mythril are general-purpose analyzers for common vulnerabilities in smart contracts, for fair comparison, we only enable the relevant queries in the evaluation.

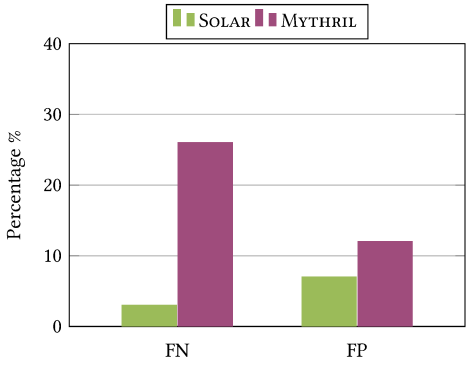

Figure 9: Comparing Solar **against** Mythril

Mythril and Solar on the Etherscan data set, and both systems use a timeout of 10 minutes.

Summary of results. For 156 contracts flagged as Reentrancy vulnerablity by at least one tool, we manually determine the ground truth and summarize the results in Figure 9. The false negative (FN) and false positive (FP) rates of Solar are 7% and 3%, while the FN and FP rates of Mythril are 26% and 12%.

Performance. Mythril takes an average of 23 seconds to analyze a contract, while Solar takes an average of 8 seconds for this data set.

Discussion. The high false negative rate in Mythril is caused by low coverage on the corresponding benchmarks. In the presence of large and complex methods, Mythril fails to generate traces that trigger the vulnerability. Moreover, Mythril does not support cross-function re-entrancy—i.e., re-entrancy attacks that span multiple functions of the victim contract.

We also investigated the cause of false positives reported by Solar. It turns out that the false positives are caused by the imprecision of our queries. In particular, we use a specific pattern of traces to *overapproximate* the behavior of the Reentrancy attack. While effective and efficient in practice, our query may generate spurious exploits that are infeasible. To mitigate this limitation, one compelling approach for developing secure smart contracts is to ask the developers to provide invariants that the tool can use to rule out infeasible attacks.

Comparison with teether**.** We next compare Solar against teether [42], the most recent tool using dynamic symbolic execution for generating exploits that would enable the attacker to control the money transactions of a victim contract. In particular, teether looks for so-called *critical instructions* (i.e., `call`, `selfdestruct`, etc.) that include recipients' addresses, which can be manipulated by the attacker to withdraw tokens from a vulnerable contract.

Summary of results. In total, there are 198 contracts that are marked as *attack-control* vulnerability by at least one tool. While Solar covers all exploits generated by teether, Solar also finds 21 *extra* exploits that cannot be generated by teether.

Performance. teether takes an average of 31 seconds to analyze a contract in the Etherscan data set, while Solar takes an average of 8 seconds per contract.

Vulnerability	Solar			ContractFuzzer		
	No.	FP	FN	No.	FP	FN
Timestamp	16	0	1	13	3	7
Gasless Send	17	0	0	14	3	6
Bad Random	9	0	0	5	1	5

Table 2: Comparing Solar **against** ContractFuzzer

Discussion. The missing exploits in teether are caused by low coverage on the corresponding benchmarks. For the 21 benchmarks with exploits that cannot be generated by teether, 14 involve attack programs with four method calls, and each of the remaining 7 benchmarks contains over 3000 lines of source code with complex control flow. As a result, teether fails to explore sufficiently many *concrete traces* to find the exploits, even if we increase the timeout from 10 minutes to 1 hour.

Comparison with ContractFuzzer**.** We further compared Solar against ContractFuzzer [39], a recent smart contract analyzer based on dynamic fuzzing. ContractFuzzer takes as input the ABI interfaces of smart contracts and *randomly* generates inputs invoking the public methods provided by the ABI. To verify the correctness of the exploits, ContractFuzzer implements oracles for different vulnerabilities by instrumenting the Ethereum Virtual Machine (EVM) with extra assertions.

We use the docker image [8] provided by the author of ContractFuzzer. The original paper does not discuss the performance of the tool, but from our experience, ContractFuzzer is slow, taking more than 10 mins to fuzz a smart contract. Since it would be time-consuming to run ContractFuzzer on the Etherscan data set, we evaluate both tools on the 33 benchmarks from the ContractFuzzer artifact [9] plus another 67 random samples from Etherscan for which we know the ground truth.

Summary of results. The results of our evaluation are summarized in Table 2. For the timestamp dependency, ContractFuzzer flags 13 benchmarks as vulnerable. However, 3 of them are false alarms, and ContractFuzzer fails to detect 7 vulnerable benchmarks. On the other hand, Solar detects most of the benchmarks with only one false negative, which is caused by a timeout of the Vandal decompiler [34].

Similarly, for the Gasless-send vulnerability, 14 benchmarks are flagged by ContractFuzzer. However, 3 of them are false positives, and 6 vulnerable benchmarks can not be detected within 10 minutes. In contrast, Solar successfully generates exploits for all the vulnerable benchmarks.

Performance. On average, ContractFuzzer takes 10 mins to analyze a smart contract. Solar takes an average of 11 seconds on this data set.

Discussion. The cause of false negatives in ContractFuzzer is easy to understand as it is based on random, rather than exhaustive, exploration of an extremely large search space. So if there are relatively few inputs in this space that lead to an attack, ContractFuzzer is unlikely to find one in reasonable time. The false positives in ContractFuzzer are caused by the limited expressiveness of its assertion language. For instance, the Time Dependency is defined

$S^\dagger$-mean	$S^\diamond$-mean	# of Benchmarks Timeout		
		$S^\dagger \wedge S^\diamond$	$S^\dagger - S^\diamond$	$S^\diamond - S^\dagger$
8s	35s	1846	548	17454

Table 3: Comparison between summary-based ($S^\dagger$) and non-summary ($S^\diamond$). $S^\dagger \wedge S^\diamond$, $S^\dagger - S^\diamond$, and $S^\diamond - S^\dagger$ represent number of benchmarks timeout on both, $S^\dagger$ only, and $S^\diamond$ only, respectively.

as the following assertion in ContractFuzzer:

TimestampOp $\wedge$ (**SendCall** $\vee$ **EtherTransfer**)

The assertion raises a Time Dependency vulnerability if the smart contract contains the `timestamp` and `call` instructions. It is easy to raise false alarms with this assertion if the `call` instruction does not depend on `timestamp`.

Result for RQ1: Solar outperforms three state-of-the-art analyzers in terms of running time, false positives, and false negatives.

7.2 Impact of Summary-based Symbolic Evaluation

To understand the impact of our summary-based symbolic evaluation described in Section 5, we use the Reentrancy vulnerability as the client and run Solar on the Etherscan data set with ($S^\dagger$) and without ($S^\diamond$) computing the summary. To speed up the evaluation, for both settings, we enable the parallel synthesis optimizations discussed in Section 6.

Figure 10 shows the results of running Solar with different settings and a time limit of 10 minutes. Each dot in the figure represents the pairwise running time of a specific benchmark under different settings; a dot near the diagonal indicates that the performance of two settings is similar. Our summary-based symbolic evaluation significantly outperforms the baseline (i.e., non-summary) in the vast majority of benchmarks. As shown in Table 3, if we exclude the benchmarks that timeout in 10 minutes, the mean time of our summary-based symbolic evaluation is only 8 seconds, while it takes 35 seconds without computing the summary. Furthermore, 1846 benchmarks time out for both settings, and only 548 benchmarks time out on $S^\dagger$ but not on $S^\diamond$. However, without computing the summary, 17454 (i.e., 69.8%) benchmarks time out. The result confirms that the summary-based technique is key to the efficiency of Solar.

Result for RQ2: Our summary-based technique is key to the efficiency of Solar.

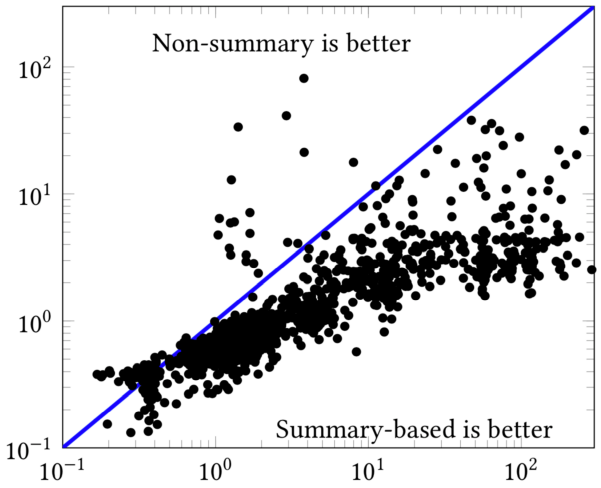

Figure 10: Comparison of run times (in seconds) between non-summary (x-axis) and summary-based (y-axis) (log-scale).

8 RELATED WORK

Smart contract security has been extensively studied in recent years. This section briefly discusses prior closely related work.

Smart Contract Analysis. Many popular security analyzers for smart contracts are based on symbolic execution [41]. Well-known tools include Oyente [43], Mythril [12] and Manticore [3]. Their key idea is to find an execution path that satisfies a given property or assertion. While Solar also uses symbolic evaluation to search for attack programs, our system differs from these tools in two ways. First, the prior tools adopt symbolic execution for *bug finding*. Our tool can be used not only for bug finding but also for *exploit generation*. Second, while symbolic execution is a powerful and precise technique for finding security vulnerabilities, it does not guarantee to explore all possible paths, which leads to false negative rates as shown in Section 7.1. In contrast, Solar analyzes all (bounded) paths through a contract using summary-based symbolic evaluation, which significantly reduces the number of paths that the underlying Rosette engine has to execute symbolically while maintaining the same precision.

To address the scalability and path explosion problems in symbolic execution, researchers developed sound and scalable static analyzers [34, 36, 40, 48]. Both Securify [48] and Madmax [34] are based on abstract interpretation [32], which soundly overapproximates and merges execution paths to avoid path explosion. The ZEUS [40] system takes the source code of a smart contract and a policy as inputs, and then compiles them into LLVM IRs that will be checked by an off-the-shelf verifier [46]. The ECF [36] system is designed to detect the DAO vulnerability. Similar to our tool, Securify also provides a query language to specify the patterns of common vulnerabilities. Unlike our tool, none of these systems can generate exploits. We could not directly compare Solar with Zeus as the tool and benchmarks are not publicly available. However, we note that our system is complementary to existing static analyzers such as Securify: in particular, we can use Securify to filter out safe smart contracts and leverage Solar to generate exploits for vulnerable ones.

Some systems [35, 38, 45] for reasoning about smart contracts rely on formal verification. These systems prove security properties of smart contracts using existing interactive theorem provers [1]. They typically offer strong guarantees that are crucial to smart contracts. However, unlike our system, all of them require significant

manual effort to encode the security properties and the semantics of smart contracts.

Automatic Exploitation. Our work is also closely related to automatic exploitation [28, 31, 39, 42]. While prior systems rely on constraint solvers to generate counterexamples as potential exploits, we note that there are additional challenges in automatic exploitation for smart contracts. First, the exploits in classical vulnerabilities (e.g., buffer overflows, SQL injections) are typically program inputs of a specific data type (e.g., integer, string) whereas the exploits in our setting are adversarial smart contracts that faithfully model the execution environment (storage, gas, etc.) of the EVM. Second, Keccak-256 hash is ubiquitous in smart contract for accessing addresses in memory or storage. As shown in Section 7.1, basic symbolic execution will fail to resolve the Keccak-256 hash, resulting in poor coverage. To address this problem, the TEETHER [42] system proposed a novel algorithm to infer the memory addresses encoded as Keccak-256 hash. Unlike TEETHER, our system directly synthesizes function calls that manipulate the memory and storage thus avoids expensive computation to resolve the hash values. Our evaluation in Section 7.1 shows that SOLAR outperforms the TEETHER tool in terms of both running time and false negatives. Similar to SOLAR, CONTRACTFUZZER [39] also generates exploits for a limited class of vulnerabilities based on the ABI specifications of smart contracts. However, as shown in Section 7.1, since CONTRACTFUZZER is based on random input generation, it is an order of magnitude slower than SOLAR, resulting in many missed exploits compared to SOLAR. Its assertion language is also less expressive than ours, leading to false positives that SOLAR avoids.

SYMBOLIC EVALUATION. SOLAR builds on the Rosette [47] symbolic evaluation engine with a new summary-based technique for scaling symbolic evaluation to large programs in the domain of smart contracts. As shown in Section 7.2, this technique is critical for performance. The idea of computing summaries to speed up symbolic evaluation has also been explored in the context of symbolic execution (see [29] for a survey), leading to three main approaches [26, 30, 33]. Two of these approaches [26, 33] compute summaries path-by-path, so a full summary that encodes all (bounded) paths through a program would be, in the worst case, exponential in program size. Prior tools therefore avoid computing full summaries, instead summarizing a subset of all paths for the purpose of test generation. SOLAR, in contrast, summarizes all (bounded) paths through a procedure, and produces compact (polynomially-sized) summaries by employing a symbolic evaluator [47] that combines symbolic execution and bounded model checking. Another summarization approach [30] uses a caching scheme that lets the underlying symbolic execution engine terminate the exploration of a path as soon as it reaches a previously seen state. The scheme does not compute explicit summaries of code; instead, it only stores enough information to soundly decide when the symbolic execution of a path reaches a previously seen state. In contrast, our approach computes an explicit and precise summary of a procedure's semantics.

PROGRAM SYNTHESIS. SOLAR uses syntax-guided synthesis [25] to search for attack programs. Synthesizers of this kind (see [37] for a survey) rely on either enumerative search (which can be stochastic or exhaustive) or symbolic reasoning or a combination of the two. SOLAR combines exhaustive enumeration with symbolic synthesis (Section 6.1), and extends this with a parallel symbolic evaluation technique (Section 6.2) for fast enumeration. Both optimizations are specialized to the domain of smart contracts, and they are critical for performance: disabling them renders the system unusable.

9 CONCLUSION

This paper presented SOLAR, a tool for automatic synthesis of adversarial contracts that exploit vulnerabilities in a victim smart contract. To make synthesis tractable, SOLAR introduces *summary-based symbolic evaluation*, which enables our tool to perform precise all-paths analysis of large real-world contracts, while significantly reducing the number of paths that need to be executed symbolically. SOLAR also introduces optimizations to partition the synthesis search space for parallel exploration. Evaluating SOLAR on the entire ETHERSCAN data set, we find that it significantly outperforms state-of-the-art analyzers in terms of precision and execution time.

ACKNOWLEDGEMENTS

This work has been supported in part by the NSF Grants CCF-1651225, ACI OAC–1535191, FMitF CCF-1918027, OIA-1936731, SaTC-1908494, by the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF-1723352), the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01, grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032, as well as gifts from Adobe, Facebook, Google, Intel, and Qualcomm.

REFERENCES

[1] 2016. The Coq Proof Assistant. https://coq.inria.fr/. [Online; accessed 01/09/2019].
[2] 2016. GovernMental's 1100 ETH payout is stuck because it uses too much gas. https://tinyurl.com/y83dn2yf/. [Online; accessed 01/09/2019].
[3] 2016. Manticore. https://github.com/trailofbits/manticore/. [Online; accessed 01/09/2019].
[4] 2017. On the parity wallet multisig hack. https://tinyurl.com/yca83zsg/. [Online; accessed 01/09/2019].
[5] 2017. The Racket Language. https://racket-lang.org/. [Online; accessed 01/09/2019].
[6] 2017. Understanding The DAO Attack. https://tinyurl.com/yc3o8ffk/. [Online; accessed 01/09/2019].
[7] 2018. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. https://ethereum.github.io/yellowpaper/paper.pdf. [Online; accessed 01/09/2019].
[8] 2018. The Ethereum Smart Contract Fuzzer for Security Vulnerability Detection. https://github.com/gongbell/ContractFuzzer. [Online; accessed 01/09/2019].
[9] 2018. The Ethereum Smart Contract Fuzzer for Security Vulnerability Detection. https://github.com/gongbell/ContractFuzzer. [Online; accessed 01/09/2019].
[10] 2018. Ethereum Smart Contract Security Best Practices. https://consensys.github.io/smart-contract-best-practices/. [Online; accessed 01/09/2019].
[11] 2018. Etherscan. https://etherscan.io/. [Online; accessed 01/09/2019].
[12] 2018. Mythril Classic. https://github.com/ConsenSys/mythril-classic. [Online; accessed 12/01/2018].
[13] 2018. New batchOverflow Bug in Multiple ERC20 Smart Contracts. https://tinyurl.com/yd78gpyt. [Online; accessed 01/09/2019].
[14] 2018. Parity Multisig Wallet Hacked, or How Come? https://cointelegraph.com/news/parity-multisig-wallet-hacked-or-how-come. [Online; accessed 01/09/2019].
[15] 2018. Real Estate Business Integrates Smart Contracts. https://tinyurl.com/yawrkfpx/. [Online; accessed 01/09/2019].
[16] 2018. Smart contracts for shipping offer shortcut. https://tinyurl.com/yavel7xe/. [Online; accessed 01/09/2019].
[17] 2018. Time manipulation. https://dasp.co/. [Online; accessed 01/09/2019].
[18] 2018. Unchecked Return Values For Low Level Calls. https://dasp.co. [Online; accessed 01/09/2019].

[19] 2019. Bitcoin. https://bitcoin.org/. [Online; accessed 01/09/2019].

[20] 2019. Ethereum. https://www.ethereum.org/. [Online; accessed 01/09/2019].

[21] 2019. Ethereum Yellow Paper. https://github.com/ethereum/yellowpaper. [Online; accessed 01/09/2019].

[22] 2019. Serpent. https://github.com/ethereum/serpent. [Online; accessed 01/09/2019].

[23] 2019. Solidity. https://solidity.readthedocs.io/en/v0.5.1/. [Online; accessed 01/09/2019].

[24] 2019. Vyper. https://github.com/ethereum/vyper. [Online; accessed 01/09/2019].

[25] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. 1–25.

[26] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 367–381. https://doi.org/10.1007/978-3-540-78800-3_28

[27] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 164–186.

[28] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Proc. The Network and Distributed System Security Symposium*.

[29] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[30] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 351–366. https://doi.org/10.1007/978-3-540-78800-3_27

[31] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc. IEEE Symposium on Security and Privacy*. 380–394.

[32] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Symposium on Principles of Programming Languages*. 238–252.

[33] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. 47–54. https://doi.org/10.1145/1190216.1190226

[34] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 116:1–116:27.

[35] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 243–269.

[36] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. In *Proc. Symposium on Principles of Programming Languages*. 48:1–48:28.

[37] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2, 1–119.

[38] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*. 520–535.

[39] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proc. International Conference on Automated Software Engineering*. 259–269.

[40] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proc. The Network and Distributed System Security Symposium*.

[41] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[42] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proc. USENIX Security Symposium*. 1317–1333.

[43] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proc. Conference on Computer and Communications Security*. 254–269.

[44] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014 (published 2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), 53–58.

[45] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Rosu. 2018. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 912–915.

[46] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proc. International Conference on Computer Aided Verification*. 106–113.

[47] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *Proc. Conference on Programming Language Design and Implementation*. 530–541.

[48] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proc. Conference on Computer and Communications Security*. 67–82.