Precise and Scalable Attack Synthesis for Smart Contracts

No Author Given

No Institute Given

Abstract. Smart contracts are programs running on top of blockchain platforms. They interact with each other through well-defined interfaces to perform financial transactions in a distributed system with no trusted third parties. But these interfaces also provide a favorable setting for attackers, who can exploit security vulnerabilities in smart contracts.
This paper presents SOLAR, a system for automatic synthesis of adversarial contracts that identify and exploit vulnerabilities in a victim smart contract. Our tool explores the space of *attack programs* based on the Application Binary Interface (ABI) specification of a victim smart contract in the Ethereum ecosystem. To make the synthesis tractable, we introduce *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. Building on the summary-based symbolic evaluation, SOLAR further introduces an approach for partitioning the synthesis search space for parallel exploration, as well as a lightweight deduction technique that can prune infeasible candidates earlier. We encoded common vulnerabilities of smart contracts in our query language, 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 CFUZZER, in terms of running time, precision, and soundness. Furthermore, running on the ETHERSCAN contracts, SOLAR synthesizes 9 previously unknown attacks that exploit the recent BATCHOVERFLOW vulnerability and that are missed by existing tools.

1 Introduction

Smart contracts are programs running on top of blockchain platforms such as Bitcoin [22] and Ethereum [23]. 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 [26]), 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 [13] instances covering financial products, online gaming, real estate [18], shipping, and logistics [19].

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 [6, 4, 2, 30]. 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 *precisely* model the inter-contracts communication and reason about the execution in a *precise* and *scalable* way. But existing tools either soundly *overapproximate* the execution a smart contract and report warnings [50, 37] that do not correspond to feasible paths and therefore cannot be exploited, or they precisely enumerate [45, 42, 46] *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 between expressiveness 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 engine employs a novel summary-based symbolic evaluation technique to scale precise reasoning to large contracts that are out of reach of existing symbolic execution [45, 46] and fuzzing [42] tools. Previous summarization techniques [29, 36] rely on symbolic execution and can therefore lead to summaries that are exponential in program size. Our technique relies on Rosette [49], 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.

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

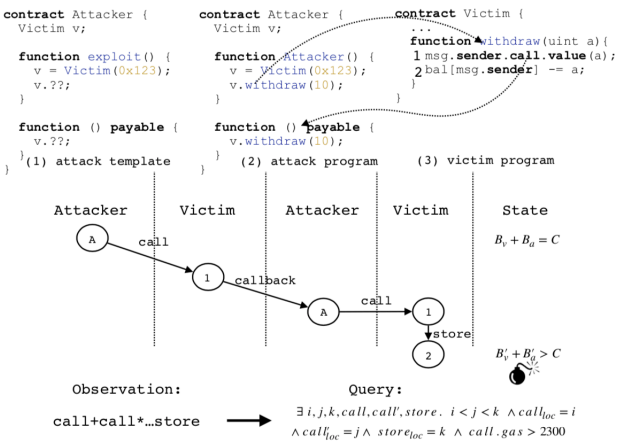

Fig. 1. Sample contracts to show the Reentrancy attack.

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 [49]. SOLAR further reduces the search space by pruning infeasible candidates early, using their symbolic encoding to quickly check for the absence of potentially exploitable paths. After that, our tool symbolically evaluates each remaining candidate to check if any of them satisfies the vulnerability query. If so, the candidate is returned as a potential exploit.

We have evaluated SOLAR on the entire data set (>25K) from ETHERSCAN [13], 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 [20], and malicious access control [45]), Security Best Practices [11], as well as the recent BATCHOVERFLOW Bug [15] (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 [45] and two orders of magnitude faster than CFUZZER [42]. SOLAR is also effective in that it significantly outperforms state-of-the-art smart contracts analyzers, namely, TEETHER, MYTHRIL, and CFUZZER, in terms of false positive and false negative rates. Furthermore, running on the ETHERSCAN contracts, SOLAR synthesizes 9 previously unknown attacks that exploit the recent BATCHOVERFLOW vulnerability and that are missed by existing tools.

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, parallel symbolic execution, and early pruning 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, precision, and soundness (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 [25], Vyper [27], and Solidity [26], and then compiled to the Ethereum Virtual Machine (EVM) bytecode [24], a low-level stack-based language. For instance, Figure 1 shows two smart contracts written in the Solidity programming language [26].

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 we 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 `call` instructions followed by a `store` operation, which can be expressed using the first-order formula in Figure 1.

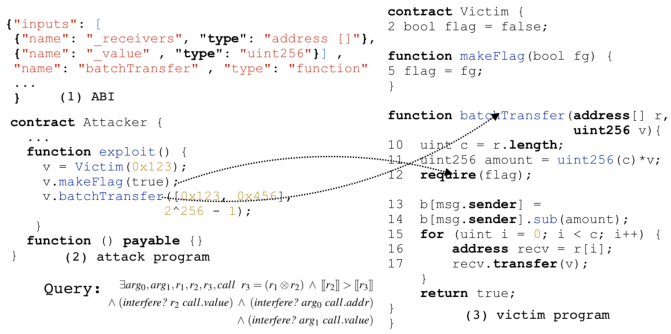

Fig. 2. An example to show the BATCHOVERFLOW attack.

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 [14, 46, 45] to generate exploits for simple properties such as attack-control [45]) in a *single contract*. But for complex vulnerabilities that require reasoning about interactions among multiple contracts (e.g., attacker versus victim in REENTRANCY

[1] Ethereum's gas mechanism ensures that this callback loop terminates.

or caller versus callee in Parity Multisig [17]), existing tools provide either no support [45] or very limited support that leads to high rates [46] of false positives and negatives (as shown in Section 7.2). Yet Alice can easily initialize the boilerplate code for basic interactions, like the "attack template" on the left hand side of Figure 1. 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) [15] 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 12. Next, it needs to assign a large number to `v` that leads to an overflow at line 11. Finally, it specifies the attacker's address as the beneficiary of the transaction (line 17). 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 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 three additional optimizations. First, instead of exploring the space of concrete programs,

we leverage ROSETTE [49] to partition this space into a small set of *symbolic programs* (Section 6.1). Second, instead of eagerly exploring the space of symbolic programs, we design a simple but effective *early pruning* strategy that allows SOLAR to prune *infeasible* symbolic candidates before executing them (Section 6.3). Finally, 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).

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: `call` transfers the balance from the current contract to a recipient whose address is specified as the argument, `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 [47].

We define the operational semantics of each statement in Figure 3 based on the standard defined by the EVM yellow paper [8]. The semantics is lifted to work on symbolic values in the standard way [49]. 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 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 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'*.*

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

Fig. 3. Intermediate language for smart contract

(a) Solidity program

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

(b) Symbolic evaluation

```
assert(_amount > 0);
r1 := _amount - 1;
sstore(vesting.amount, _amount - 1);
call(msg.sender, _to, _amount - 1);
r2 := amount - 15;
r3 := amount - 15;
r4 := sload(vesting.startTime);
no-op;
```

(c) Summary extraction

$\overline{sstore}(\texttt{vesting.amount}, \Gamma_S[\texttt{\_amount}] - 1)@(\Gamma_S[\texttt{\_amount}] > 0)$;
$\overline{call}(\Gamma_S[\texttt{msg.sender}], \Gamma_S[\texttt{\_to}], \Gamma_S[\texttt{\_amount}] - 1)@(\Gamma_S[\texttt{\_amount}] > 0)$;

(d) Summary interpretation

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

Fig. 4. From Standard to Summary-Based Symbolic Evaluation

Example 1. 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 $\Gamma[\texttt{\_amount}] - 1$. Since SOLAR does not model the event system in Solidity, we turn the corresponding instructions (e.g., line 7 in Figure 4b) into `no-op`s.

4.2 Smart Contract Vulnerabilities

Figure 5 shows our vulnerability query language. Here, `arg`, `reg`, `mem`, and `store` are variables from function arguments, registers, memory, and storage, respectively. Furthermore, variables can also refer to global properties (e.g., balance, address, timestamp) of a smart contract. We use $[\![\text{var}]\!]$ to denote the concrete or symbolic value held by `var`. The predicate `(interfere? var e)` determines whether `var` can interfere with `e`, as specified in Definition 4. The expression `(inst opcode var var ...)` represents an instruction. For instance, `(inst call` v_1 v_2 v_3 l`)` denotes a call instruction at location l where variables v_1, v_2, and v_3 represent operands that hold the gas, address, and value of the call. With slight abuse of notation, we will use `inst.operand` to refer to the operand of an instruction `inst`. For instance, in the previous instruction, the symbolic expression held by v_1 can be referenced as *call*.gas. Finally, we can express more complex queries by composing simple expressions with logical operators ($\neq, \vee, \wedge, \exists$, etc.). For queries that contain quantifiers, we use skolemization to make them quantifier-free (or reject them if they cannot be skolemized).

$\langle var \rangle$ ::= $\langle arg \rangle$ | $\langle reg \rangle$ | $\langle mem \rangle$ | $\langle store \rangle$ | timestamp | ...
$\langle opcode \rangle$ ::= call | jmp | store | ...
$\langle E \rangle$ ::= $\langle const \rangle$ | $[\![\text{var}]\!]$ | $\langle var \rangle$ | $\langle E \rangle \oplus \langle E \rangle$ | $\neg\langle E \rangle$ | $\forall\langle var \rangle.\ \langle E \rangle$ | $\exists\langle var \rangle.\ \langle E \rangle$ | (interfere? $\langle var \rangle$ $\langle E \rangle$)
| (inst $\langle opcode \rangle$ $\langle var \rangle$ $\langle var \rangle$ ...)
($\oplus \in \{+, -, >, =, \neq, \vee, \wedge, ...\}$)

Fig. 5. Query language for SOLAR

Definition 3. ***(Vulnerability)*** *A* Vulnerability $\mathcal{V}$ *is a predicate over a set of variables* V *in the program state. A vulnerability* $\mathcal{V}$ *appears in the program* P *if the execution of* P *can reach a program state* Γ' *that satisfies* $\mathcal{V}$*:* $\Gamma' \models \mathcal{V}$*.*

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

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

$$\begin{aligned} \exists arg_0, arg_1, r_1, r_2, r_3, call \ \ & r_3 = (r_1 \otimes r_2) \ \wedge \ [\![r_2]\!] > [\![r_3]\!] \\ \wedge\ (\mathit{interfere?}\ r_2\ \mathit{call.value}) \ \wedge \ & (\mathit{interfere?}\ arg_0\ \mathit{call.addr}) \\ & \wedge\ (\mathit{interfere?}\ arg_1\ \mathit{call.value}) \end{aligned}$$

In other words, the victim program contains a `call` 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.

A Timestamp Dependency vulnerability occurs if a transaction depends on a timestamp:

Vulnerability 2 Timestamp Dependency

$$\exists\ \mathit{timestamp}, \mathit{call}.\ \mathit{call.value} > 0\ \wedge (\mathit{interfere?}\ \mathit{timestamp}\ \mathit{call.value})$$

This vulnerability enables a malicious miner to gain an advantage by choosing a suitable timestamp for a block.

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 [21]. We formalize the absence of this check as follows:

Vulnerability 3 Unchecked-send (Gasless-send)

$$\neg\forall\ \mathit{call}, \exists\ \mathit{jmp}\ \ (\mathit{interfere?}\ \mathit{call.ret}\ \mathit{jmp.var})$$

Here, the return value of a `call` 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 4 Reentrancy

$$\exists\ arg, i, j, k, call, call', store.\ \ i < j < k\ \ \wedge call_{loc} = i \wedge call'_{loc} = j$$
$$\wedge\ store_{loc} = k\ \ \wedge\ \ call.gas > 2300 \wedge (interfere?\ arg\ call.addr)$$

In other words, if an attack program has the minimum gas (i.e., 2300) to control the recipient of a transaction and generate consecutive `call` instructions before updating the storage, 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 $\mathcal{C}$ corresponds to a public method provided by the victim contract. We use Υ to denote the union of all publicly available methods.

Definition 5. *(***Component)** *A* Component $\mathcal{C}$ *from an ABI configuration is a pair* (f, τ) *where: 1)* f *is* $\mathcal{C}$*'s name, and 2)* τ *is the type signature of* $\mathcal{C}$*.*

Example 2. 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 6. *(***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*

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

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

Definition 7. *(***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 3. 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);
```

The `choose` construct is a notational shorthand for a conditional statement that guards the specified choices with fresh symbolic booleans. For example, `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 8. *(***Problem Specification)** *The specification for our* attack synthesis *problem is a tuple (Γ_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 9. *(***Attack Synthesis)** *Given a specification (Γ_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 $\exists \boldsymbol{v}.\mathcal{V}([\![\mathcal{S}]\!]_{\Gamma_0})$, where $\boldsymbol{v}$ denotes the symbolic variables in $\mathcal{S}$. A model of this formula, if it exists, binds every variable in $\boldsymbol{v}$ 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 [49]. 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 10. *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 of executing s.*

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.[2] 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 [49].

Example 4. Figure 4c shows the summary of the program in Figure 4b. Using the rule in Figure 6, our tool summarizes the side effects of the `call` and `sstore` instructions at lines 2 and 3, respectively. The remaining instructions are omitted from the summary because they have no persistent side effects.

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

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

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.

Example 5. 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 `call` 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 `call` statement in the environment Γ.

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

Fig. 6. Procedure for summary generation.

```
1  (define (interpret-summary s@φ Γ)
2    (define s_Γ@φ_Γ (substitute s@φ Γ))
3    (match s_Γ
4      [call(x_Γ, y_Γ, z_Γ) (when φ_Γ call(x_Γ, y_Γ, z_Γ))]
5      [sstore(x, y_Γ) (when φ_Γ sstore(x, y_Γ))]
6      [_  no-op]))
```

Fig. 7. Procedure for summary interpretation

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 [49] 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.

```
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))
```

Fig. 8. Solar implementation in Rosette.

Figure 8 shows the implementation of Solar in Rosette. The tool takes as input a vulnerability specification $\mathcal{V}$, the components Υ of a 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 [8]. 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 [37] 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 [35]. 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 [24].

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 6) 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:

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

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.[3] 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.

6.3 SMT-based Early Pruning

In addition to hoisting, we also design a simple but effective *early pruning* strategy that allows SOLAR to prune *infeasible* symbolic programs before executing them. The intuition behind our strategy is that all attacks expressible in SOLAR (e.g., [6, 4, 2]) invoke at least one public method that manipulates persistent storage and at least one public method that transfers cryptocurrency using the `call` instruction. In other words, a successful attack executes at least one store instruction followed by at least one `call` instruction. We express our early pruning strategy using the following ROSETTE program:

```
1 (define (may-store-and-call? p)
2  (solve (exists (list i j)
3   (and (< i j) (= (type p[i]) 'store)
4 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(=␣(type␣p[j])␣'call)))))
```

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

This procedure queries the solver to find out if the given symbolic program p contains any concrete attack program that executes a `call` after a store. This query is much faster to solve than a vulnerability query, so if p contains no feasible candidate, SOLAR does not run the vulnerability query for it.

7 Evaluation

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

- Q1: *Effectiveness*: How does SOLAR compare against state-of-the-art analyzers for smart contracts?
- Q2: *Efficiency*: How much does summary-based symbolic evaluation improve the performance of SOLAR?
- Q3: *Expressiveness*: Can SOLAR express the specifications of recent vulnerabilities?

To answer these questions, we perform a systematic evaluation by running SOLAR on the entire set of smart contracts from ETHERSCAN [13]. Using a snapshot from August 30 2018, we obtained a total number of 25,983 smart contracts. 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 CFUZZER, based on dynamic random testing.

7.2 Expressiveness of SOLAR

To understand the expressiveness of our tool, we encoded the common vulnerabilities in smart contracts described in prior work [20, 12] and on social media [7]. In particular, Table 1 summarizes the expressiveness of mainstream tools for smart contract security, ordered by publish date. Note that our tool supports not only well-known vulnerabilities such as Reentrancy, Timestamp Dependency, and Arithmetic operations (i.e., over/underflow), but also recent attacks such as the BATCHOVERFLOW vulnerability discussed in Section 3. Prior tools express a portion of these vulnerabilities. For instance, the OYENTE [46] tool, which is also based on symbolic execution, does not support vulnerabilities such as unchecked calls and out-of-gas-DoS. Static analysis tools such as Securify [50] and MADMAX [37] do not support complex arithmetic vulnerabilities. Most importantly, unlike SOLAR, none of them can generate exploits for vulnerabilities.

The TEETHER and the CFUZZER tools can automatically generate exploits, but their systems only support a small class of vulnerabilities.

There are some vulnerabilities that our tool does not support well. For instance, a Transaction-Ordering Dependency (TOD) is a race condition vulnerability, and exploiting it requires synthesizing *a pair of programs* that exhibit the race. In the future, we plan to explore *relational synthesis* to handle attacks that require multiple programs. Another source of limitation is denial-of-service (DoS) attacks that involve loops, which our tool unrolls during symbolic execution, and the unrolling bound may not be large enough to trigger the vulnerability.

Tool	Generate Exploit?	Common Vulnerabilities							
		Reentrance	Arithmetic	DoS	Bad Random	Timestamp	TOD	Unchecked Calls	Attack Control
OYENTE	◐	✓	✓			✓	◐		
Mythril	◐	✓	✓			✓	◐	✓	✓
Zeus		✓	✓			✓	◐	✓	✓
TEETHER	✓								✓
Securify		✓				✓	◐	✓	✓
MADMAX			◐	◐					
CFUZZER	✓	✓			✓	✓	◐		
SOLAR	✓	✓	✓	◐	✓	✓	◐	✓	✓

Table 1. A Comparison of Existing Tools (OYENTE [46], Mythril [14], Zeus [43], TEETHER [45], Securify [50], MADMAX [37], and ContractFuzzer [42]) for Smart Contract (Order by publish date). ◐ represents limited support.

.

Comparison with MYTHRIL We first compare with MYTHRIL [14] 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 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 vulnerable 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 span over multiple functions of the victim contract.

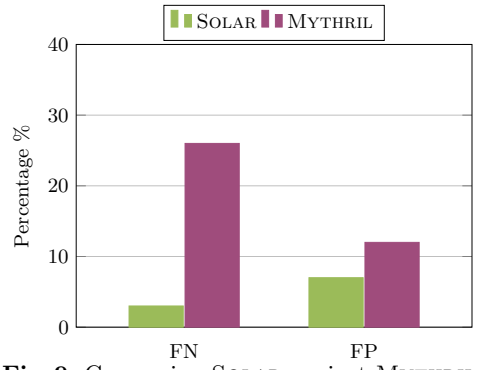

Fig. 9. Comparing Solar against Mythril

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 [45], 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 vulnerable 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 the Etherscan data set, while Solar takes an average of 8 seconds per contract.

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 CFuzzer We further compared Solar against CFuzzer [42], a recent smart contract analyzer based on dynamic fuzzing. CFuzzer 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, CFuzzer implements oracles for different vulnerabilities by instrumenting the Ethereum Virtual Machine (EVM) with extra assertions.

Vulnerability	Solar			CFuzzer		
	No.	FP	FN	No.	FP	FN
Timestamp	16	0	1	13	4	7
Gasless Send	17	0	0	14	3	6
Bad Random	9	0	0	5	1	5

Table 2. Comparing Solar against CFuzzer

We use the docker image [9] provided by the author of CFuzzer. The original paper does not discuss the performance of the tool, but from our experience, CFuzzer is slow, taking more than 10 mins to fuzz a smart contract. Since it would be time-consuming to run CFuzzer on the Etherscan data set, we evaluate both tools on the 33 benchmarks from the CFuzzer artifact [10] 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, CFuzzer flags 13 benchmarks as vulnerable. However, 4 of them are false alarms, and CFuzzer 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 [37].

Similarly, for the Gasless-send vulnerability, 14 benchmarks are flagged by CFuzzer. 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, CFuzzer 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 CFuzzer 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, CFuzzer is unlikely to find one in reasonable time. The false positives in CFuzzer are caused by the limited expressiveness of its assertion language. For instance, the Time Dependency is defined as the following assertion in CFuzzer:

$$\textbf{TimestampOp} \wedge (\textbf{SendCall} \vee \textbf{EtherTransfer})$$

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`. On the other hand, the `interfere?` function enables SOLAR to reason about this dependency precisely.

7.3 Impact of Summary-based Symbolic Evaluation

$S^\dagger$-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.

To understand the impact of our summary-based symbolic evaluation described in Section 5, we 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 early pruning and parallel synthesis optimizations discussed in Section 6.

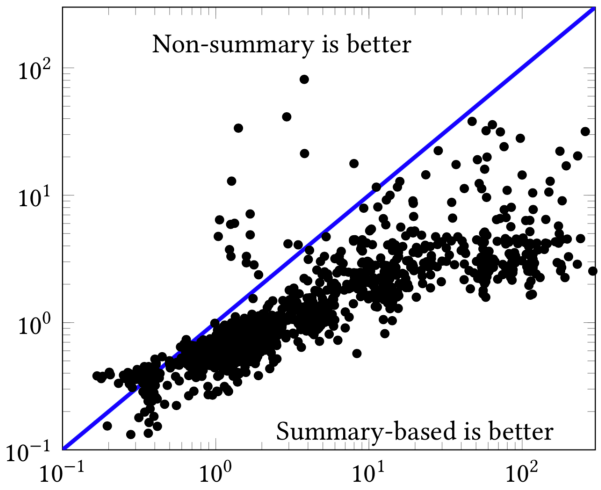

Fig. 10. Comparison of run times (in seconds) between non-summary (x-axis) and summary-based (y-axis) (log-scale).

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.

7.4 A case study on the BatchOverflow vulnerability

To evaluate whether Solar can express and discover new vulnerabilities in real world smart contracts, we conduct a case study on the recent BatchOverflow vulnerability. Exploits due to this vulnerability have resulted in the creation of trillions of invalid Ethereum Tokens in 2018 [7], causing major exchanges to temporary halt until all tokens could be reassessed. Note that generating exploits for this vulnerability is quite challenging as it requires the tool to reason about the combination of arithmetic operations, interference, and the read-write semantics of the storage system in Solidity.

Similar to our previous experiment, we first encode the BatchOverflow vulnerability (Section 4.2) in our query language and then run our tool on the Etherscan data set. In total, Solar flags 16 vulnerable contracts. To verify that the exploits are effective, we setup a private blockchain using the Geth [16] framework where we can run exploits on the vulnerable contracts. We confirmed that 9 exploits are valid. The infeasible attacks come from the incompleteness of the query as well as imprecise control flow graphs from the Vandal decompiler. Running teether on these 9 vulnerable contracts, we find that it fails to generate their exploits.

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 [44]. Well-known tools include Oyente [46], Mythril [14] 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 7.2. 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 [39, 50, 37, 43]. Both Securify [50] and Madmax [37] are based on abstract interpretation [35], which soundly overapproximates and merges execution paths to avoid path explosion. The ZEUS [43] 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 [48]. The ECF [39] 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 [41, 38] 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 [31, 34, 45, 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 7.2, basic symbolic execution will fail to resolve the Keccak-256 hash, resulting in poor coverage. To address this problem, the TEETHER [45] 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 7.2 shows that SOLAR outperforms the TEETHER tool in terms of both running time and false negatives. Similar to SOLAR, CFUZZER [42] also generates exploits for a limited class of vulnerabilities based on the ABI specifications of smart contracts. However, as shown in Section 7.2, since CFUZZER 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 [49] 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 7.3, 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 [32] for a survey), leading to three main approaches [29, 36, 33]. Two of these approaches [36, 29] 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 [49] that combines symbolic execution and bounded model checking. Another summarization approach [33] 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 [28] to search for attack programs. Synthesizers of this kind (see [40] 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, and to prune infeasible attack candidates earlier. Evaluating SOLAR on the entire ETHERSCAN data set, we find that it significantly outperforms state-of-the-art analyzers in terms of precision, soundness, and execution time. SOLAR also uncovers 9 previously unknown exploits of the BATCHOVERFLOW vulnerability that are missed by all prior tools.

References

1. The coq proof assistant. https://coq.inria.fr/ (2016), [Online; accessed 01/09/2019]
2. Governmental's 1100 eth payout is stuck because it uses too much gas. https://tinyurl.com/y83dn2yf/ (2016), [Online; accessed 01/09/2019]
3. Manticore. https://github.com/trailofbits/manticore/ (2016), [Online; accessed 01/09/2019]
4. On the parity wallet multisig hack. https://tinyurl.com/yca83zsg/ (2017), [Online; accessed 01/09/2019]
5. The racket language. https://racket-lang.org/ (2017), [Online; accessed 01/09/2019]
6. Understanding the dao attack. https://tinyurl.com/yc3o8ffk/ (2017), [Online; accessed 01/09/2019]
7. Batchoverflow exploit creates trillions of ethereum tokens, major exchanges halt erc20 deposits. https://tinyurl.com/yb6eo6r9 (2018), [Online; accessed 01/09/2019]
8. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf (2018), [Online; accessed 01/09/2019]
9. The ethereum smart contract fuzzer for security vulnerability detection. https://github.com/gongbell/ContractFuzzer (2018), [Online; accessed 01/09/2019]
10. The ethereum smart contract fuzzer for security vulnerability detection. https://github.com/gongbell/ContractFuzzer (2018), [Online; accessed 01/09/2019]
11. Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/ (2018), [Online; accessed 01/09/2019]
12. Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/ (2018), [Online; accessed 01/09/2019]
13. Etherscan. https://etherscan.io/ (2018), [Online; accessed 01/09/2019]
14. Mythril classic. https://github.com/ConsenSys/mythril-classic (2018), [Online; accessed 12/01/2018]
15. New batchoverflow bug in multiple erc20 smart contracts. https://tinyurl.com/yd78gpyt (2018), [Online; accessed 01/09/2019]
16. Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum (2018), [Online; accessed 01/09/2019]
17. Parity multisig wallet hacked, or how come? https://cointelegraph.com/news/parity-multisig-wallet-hacked-or-how-come (2018), [Online; accessed 01/09/2019]
18. Real estate business integrates smart contracts. https://tinyurl.com/yawrkfpx/ (2018), [Online; accessed 01/09/2019]
19. Smart contracts for shipping offer shortcut. https://tinyurl.com/yavel7xe/ (2018), [Online; accessed 01/09/2019]
20. Time manipulation. https://dasp.co/ (2018), [Online; accessed 01/09/2019]
21. Unchecked return values for low level calls. https://dasp.co (2018), [Online; accessed 01/09/2019]
22. Bitcoin. https://bitcoin.org/ (2019), [Online; accessed 01/09/2019]
23. Ethereum. https://www.ethereum.org/ (2019), [Online; accessed 01/09/2019]

24. Ethereum yellow paper. https://github.com/ethereum/yellowpaper (2019), [Online; accessed 01/09/2019]
25. Serpent. https://github.com/ethereum/serpent (2019), [Online; accessed 01/09/2019]
26. Solidity. https://solidity.readthedocs.io/en/v0.5.1/ (2019), [Online; accessed 01/09/2019]
27. Vyper. https://github.com/ethereum/vyper (2019), [Online; accessed 01/09/2019]
28. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, pp. 1–25 (2015)
29. Anand, S., Godefroid, P., Tillmann, N.: 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. pp. 367–381 (2008). https://doi.org/10.1007/978-3-540-78800-3_28, https://doi.org/10.1007/978-3-540-78800-3_28
30. Atzei, N., Bartoletti, M., Cimoli, T.: 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. pp. 164–186 (2017)
31. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: AEG: automatic exploit generation. In: Proc. The Network and Distributed System Security Symposium (2011)
32. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). https://doi.org/10.1145/3182657, https://doi.org/10.1145/3182657
33. Boonstoppel, P., Cadar, C., Engler, D.R.: 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. pp. 351–366 (2008). https://doi.org/10.1007/978-3-540-78800-3_27, https://doi.org/10.1007/978-3-540-78800-3_27
34. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Proc. IEEE Symposium on Security and Privacy. pp. 380–394 (2012)
35. Cousot, P., Cousot, R.: 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. pp. 238–252 (1977)
36. Godefroid, P.: 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. pp. 47–54 (2007). https://doi.org/10.1145/1190216.1190226, https://doi.org/10.1145/1190216.1190226
37. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. In: Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 116:1–116:27 (2018)

38. Grishchenko, I., Maffei, M., Schneidewind, C.: 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. pp. 243–269 (2018)
39. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. In: Proc. Symposium on Principles of Programming Languages. pp. 48:1–48:28 (2018)
40. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. vol. 4, pp. 1–119 (2017)
41. Hirai, Y.: 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. pp. 520–535 (2017)
42. Jiang, B., Liu, Y., Chan, W.K.: Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proc. International Conference on Automated Software Engineering. pp. 259–269 (2018)
43. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Proc. The Network and Distributed System Security Symposium (2018)
44. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
45. Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: Proc. USENIX Security Symposium. pp. 1317–1333 (2018)
46. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proc. Conference on Computer and Communications Security. pp. 254–269 (2016)
47. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation **9**, 53–58 (2014 (published 2015))
48. Rakamaric, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Proc. International Conference on Computer Aided Verification. pp. 106–113 (2014)
49. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proc. Conference on Programming Language Design and Implementation. pp. 530–541 (2014)
50. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: Proc. Conference on Computer and Communications Security. pp. 67–82 (2018)