VerX: Safety Verification of Smart Contracts

Anton Permenev[1,2] Dimitar Dimitrov[2] Petar Tsankov[1,2] Dana Drachsler-Cohen[2] Martin Vechev[2]

[1]ChainSecurity AG, Switzerland
{firstname}@chainsecurity.com

[2]ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

https://verx.ch

***Abstract*—We present VERX, the first automated verifier able to prove functional properties of Ethereum smart contracts. VERX addresses an important problem as all real-world contracts must satisfy custom functional specifications.**

VERX is based on a careful combination of three techniques, enabling it to automatically verify temporal properties of infinite-state smart contracts: (i) reduction of temporal property verification to reachability checking, (ii) a new symbolic execution engine for the Ethereum Virtual Machine that is precise and efficient for a practical fragment of Ethereum contracts, and (iii) delayed predicate abstraction which uses symbolic execution during transactions and abstraction at transaction boundaries.

Our extensive experimental evaluation on 83 temporal properties and 12 real-world projects, including popular crowdsales and libraries, demonstrates that VERX is practically effective.

***Keywords*-smart contracts, temporal specification, automated verification**

I. INTRODUCTION

Ensuring correctness of smart contracts, programs that run on top of blockchains, is a pressing security concern. Today, billions worth of USD are controlled by smart contracts, and only in the past couple of years, millions of these have been lost by exploiting subtle flaws in the logic of these programs. The issue is exacerbated as the code becomes immutable once placed on the blockchain and hence bugs found after deployment cannot be fixed.

Current audit practice. To mitigate this problem, current audit practices of smart contracts involve checking whether the code is safe against two kinds of vulnerabilities: (i) generic security errors such as reentrancy and overflows, typically achieved by running automated security tools (e.g., Securify [65], Slither [4], and Mythril [50]), and (ii) deeper, custom functional requirements, often done by manual, best-effort code inspection. An example of such a requirement is: *"The sum of deposits never exceeds the contract's balance."*. While substantial advances in creating security tools that discover generic errors were made over the last few years, there has been little progress in automating the verification of deeper, more challenging functional properties. This is particularly problematic because manually checking the satisfaction of such deeper requirements often involves non-trivial reasoning, increasing the chance that critical bugs slip in production.

Goal: Formal guarantees for smart contracts. We believe that smart contracts, similarly to any safety-critical system (e.g., controllers deployed in cars and airplanes), must be formally verified before deployment. While this observation is not new, only a handful of smart contract projects (e.g., MakerDAO [7]) have been formally verified so far. Current verification efforts are conducted using heavyweight interactive-theorem provers, such as Isabelle/HOL [9] and Coq [68]. These require non-trivial manual effort and expertise, making the audit process expensive and time-consuming, resulting in limited adoption by the developer and audit communities.

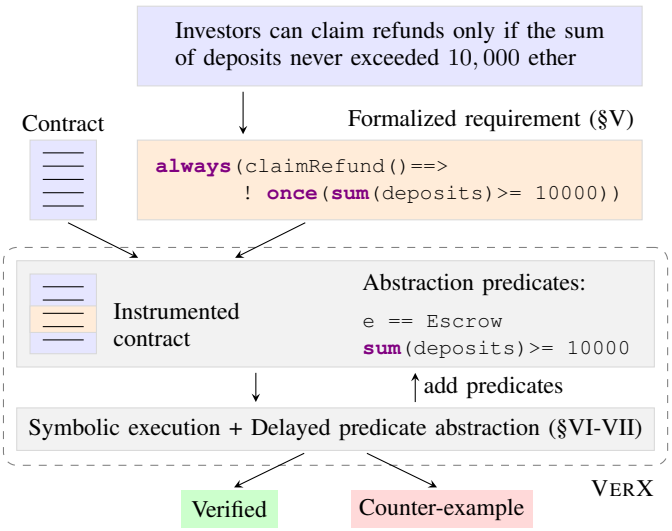

Fig. 1: Usage and verification flow of VERX. A requirement is formalized as a temporal safety property. Then, the contract is instrumented with the property and relevant predicates for verification are automatically extracted. These are fed into VERX, which either verifies the property, outputs a counter-example, or indicates that additional predicates are needed.

Key challenges. To address this problem and enable developers and auditors to formally certify smart contracts without requiring deep expertise in formal verification, one would ideally create a verifier capable of automatically proving custom, functional properties. However, building such an automated verifier is challenging for at least two reasons.

First, smart contracts often interact with external contracts via function calls. In turn, these external contracts may call back the smart contracts we want to verify in arbitrary ways (since their code is unknown). Automated verification in the presence of such arbitrary and unboundedly many callbacks from external contracts is challenging.

DOI 10.1109/SP40000.2020.00024

Second, smart contracts process an unbounded number of transactions. Therefore, even though smart contracts are usually loop-free, the verifier needs to soundly handle loops because the contracts' functions are executed in an implicit infinite loop, processing a single transaction in each iteration.

VERX: Automated verifier for Ethereum contracts. In this work, we introduce VERX, an automated verifier of functional requirements for Ethereum smart contracts, which aims to address the above challenges. The design decisions underpinning VERX were strongly motivated by the practical challenges that arise when auditing real-world smart contracts.

One of our key observations is that most real-world contracts defend against external callbacks by ensuring that these do not introduce *new* behaviors, i.e., any behavior with external callbacks is equivalent to another behavior without external callbacks; we call these *effectively external callback free* (EECF) contracts, adapting the earlier notion of effectively callback free contracts [38]. VERX focuses on verifying EECF contracts as they enjoy two important benefits. First, they simplify the formalization of requirements, as auditors can write the specification without explicitly considering all possible external callbacks. Second, the verifier can soundly avoid exploring all possible external callbacks, which facilitates precise and scalable analysis.

In Fig. 1 we depict the high-level steps performed by VERX. As a first step, we formalize the requirements as temporal safety properties in the specification language of VERX. We identify temporal properties as a particularly good fit for capturing contract requirements (also observed by [59]), because contracts process sequences of transactions and temporal properties can specify which sequences of contract states are considered valid. To ease adoption, we use the syntax of Solidity [6] (the most widely used language for Ethereum contracts) and extend it with temporal operators (e.g., `always` and `once`) so to enable quantifying over contracts' states. Fig. 1 shows the formalization of an example requirement for crowdsale contracts. In our evaluation, we illustrate further examples and present common idioms (e.g., access control, state transitions, and multi-contract invariants) of real-world contracts that have been verified using VERX.

Once the property is formalized, VERX reduces the problem of temporal safety verification to that of reachability checking by instrumenting the provided contract with the formalized property. By working with the instrumented instead of the original contract, we can directly leverage state-of-the-art program analysis methods, which typically focus on reachability verification. Such a reduction approach to verification has been shown effective in other settings (e.g., [26], [28]).

Given the instrumented contract, VERX carries out the verification by employing a particular combination of abstraction and symbolic execution, one that is especially effective in the setting of smart contracts. We refer to this combination as *delayed* predicate abstraction. Concretely, to deal with the unbounded number of transactions processed by the contract, VERX abstracts contract states which arise *in-between* transactions via a set of abstraction predicates while leveraging symbolic execution for precise analysis *within* transactions. As we demonstrate with examples later, delaying abstraction until the end of transactions is essential as key invariants often break in the middle of transactions. Moreover, precise reasoning of individual transactions using symbolic execution is possible because contract functions usually have bounded loops and recursion (so to avoid infinite computations on the blockchain).

VERX automatically infers the abstraction predicates from (i) the contract's code, such as immutable variables and pointers to contracts (e.g., `e == Escrow`), and (ii) from the property by extracting atomic predicates (e.g, `sum(deposits)>= 10000`); see Fig. 1. This means that users typically do not need to provide any predicates to VERX and can use it in a push-button manner. For example, as we show in our evaluation, 77 out of 83 properties can be automatically verified using the predicates inferred by VERX, leaving only six properties where the user had to provide additional predicates (shown by the backward arrow). Further, because existing symbolic engines for Ethereum are either unsound or inefficient (or both), we introduce a new engine that precisely models non-standard features of Ethereum's Virtual Machine (such as hash-based object allocations and gas mechanics) yet scales well for a practical fragment of Solidity.

To demonstrate the effectiveness of VERX, we present an extensive evaluation over 12 real-world projects with 83 relevant temporal properties. Our results show that VERX scales to verifying real-world contracts.

Main Contributions. Our contributions are:

- A delayed predicate abstraction approach which combines symbolic execution performed during transaction execution with abstraction done in-between transactions. Delayed abstraction is key to enabling automatic verification of temporal safety properties of smart contracts.
- The first end-to-end verification system, called VERX, which can automatically verify functional specifications of real-world contracts. VERX incorporates delayed abstraction, a new symbolic execution engine which avoids the pitfalls of existing tools, as well as sound handling of key Ethereum features.
- An extensive experimental evaluation over 12 real-world projects (138 contracts) and 83 safety properties, showing that VERX can verify interesting temporal properties of real-world smart contracts, including popular crowdsales and libraries.

II. MOTIVATING EXAMPLE AND VERIFICATION CHALLENGES

We now present a decentralized crowdsale scenario together with relevant requirements and their formalization. We then discuss key challenges when verifying such properties. In the next section we present the verification flow of VERX and how it addresses these issues.

R0 Claiming a refund by an investor decreases the escrow's balance by the investor's deposit.

R1 The escrow's balance must be at least the sum of investor deposits, unless the crowdsale is declared successful.

R2 The escrow never allows the beneficiary to withdraw the investments *and* the investors to claim refunds.

R3 Investors cannot claim refunds after more than $10,000$ ether is collected.

(a) Requirements

$$\varphi_{\mathbf{R0}} \equiv \square\big(\texttt{claimRefund(address p)} \rightarrow \texttt{Escrow.balance} = \texttt{Escrow.balance}^{\bullet} - \texttt{Escrow.deposits}^{\bullet}\texttt{[p]}\big)$$

$$\varphi_{\mathbf{R1}} \equiv \square\big(\texttt{state} \neq \texttt{SUCCESS} \rightarrow \mathit{sum}(\texttt{deposits}) \leq \texttt{Escrow.balance}\big)$$

$$\varphi_{\mathbf{R2}} \equiv \square\big(\neg\big(\blacklozenge\texttt{withdraw()} \wedge \blacklozenge\texttt{claimRefund()}\big)\big)$$

$$\varphi_{\mathbf{R3}} \equiv \square\big(\texttt{claimRefund()} \rightarrow \neg\blacklozenge\big(\mathit{sum}(\texttt{deposits}) \geq \texttt{goal}\big)\big)$$

(c) Formalized requirements as temporal safety properties

```
1  contract Crowdsale {
2    Escrow escrow;
3    uint256 closeTime;
4    uint256 raised = 0;
5    uint256 goal = 10000 * 10**18;
6
7    function constructor() {
8      escrow = new Escrow(0x1234);
9      closeTime = now + 30 days;
10   }
11
12   function invest() payable {
13     require(raised < goal);
14     // fix: uncomment pre-condition below:
15     // require(now<=closeTime);
16     escrow.deposit.value(msg.value)(msg.sender);
17     raised += msg.value;
18   }
19
20   function close() {
21     require(now > closeTime || raised >= goal);
22     if (raised >= goal) {
23       escrow.close();
24     } else {
25       escrow.refund();
26     }
27   }
28 }
```

(b) Crowdsale smart contract

```
1  contract Escrow {
2    address owner, beneficiary;
3    mapping(address => uint256) deposits;
4    enum State {OPEN, SUCCESS, REFUND}
5    State state = OPEN;
6    constructor(address b) {
7      owner = msg.sender;
8      beneficiary = b;
9    }
10   modifier onlyOwner {
11     require(msg.sender == owner);
12   }
13   function close() onlyOwner {state = SUCCESS;}
14   function refund() onlyOwner {state = REFUND;}
15   function deposit(address p) onlyOwner payable {
16     deposits[p] = deposits[p] + msg.value;
17   }
18   function withdraw() {
19     require(state == SUCCESS);
20     beneficiary.transfer(this.balance);
21   }
22   function claimRefund(address p) {
23     require(state == REFUND);
24     uint256 amount = deposits[p];
25     deposits[p] = 0;
26     p.call.value(amount)();
27   }
28 }
```

(d) Escrow smart contract

Fig. 2: Crowdsale example: (a) requirements and (c) their formalization, (b) crowdsale contract and (d) escrow contract.

A. Crowdsale Example

We present a crowdsale use case, one of the most common scenarios implemented on Ethereum. The goal here is to collect $10,000$ ether within 30 days after deployment. It is considered successful if $10,000$ ether has been collected within 30 days, in which case the beneficiary can withdraw the funds. Otherwise, the crowdsale is considered failed and users are allowed to claim a refund. We summarize four key requirements in Fig. 2a showing how users (investors and the beneficiary) can transfer funds. The crowdsale implementation is given in Fig. 2, taken from the widely-used OpenZeppelin library [3], which uses an escrow contract to lock the funds invested during the crowdsale. The contracts' initial state is defined by their constructors (the escrow is created by the crowdsale's constructor, at Line 8). In the crowdsale contract, `raised` tracks the amount of collected funds, `goal` is set to $10,000$ ether (1 ether = 10^{18} wei), and `closeTime` is set to `now` (time of deployment) plus 30 days. The escrow tracks the funds invested by individual users in the mapping `deposits`.

B. Challenge 1: Specifications in the Presence of Callbacks

As a first step, we formalize the requirements of the crowdsale and escrow contracts (referred to as a *bundle* of contracts). This is, however, nontrivial because the bundle does not execute in isolation, but may also interact with external, potentially malicious contracts.

Interactions with external contracts. We illustrate a possible interaction between the escrow and external contracts in Fig. 3. To refund an investor p, the escrow executes Line 26, which calls a designated *fallback* function in p. Here, p is any account that invested in the crowdsale, and the invoked fallback function can thus have arbitrary behavior. In Fig. 3, we consider a scenario where p calls back `claimRefund`, this time passing address `q` as an argument. The second call `claimRefund(q)` is *nested* in `claimRefund(p)`. Importantly, the nested call modifies the escrow's state (changes escrow's balance) before execution control is returned to `claimRefund(p)`. This means the state at the end of `claimRefund(p)` depends on callbacks initiated by external contracts.

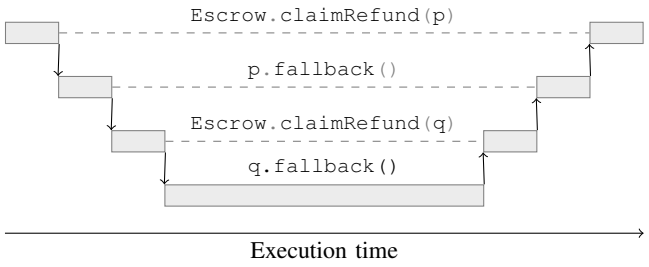

Fig. 3: Interaction between the escrow and external contracts p and q. Rectangles indicate the active function, dashed lines indicate functions waiting for external calls to return, downward and upward arrows indicate call and return points.

Specification challenge. Requirements typically implicitly assume that the bundle executes without callbacks. For example, requirement **R0** stipulates that at the end of `claimRefund(p)`, the escrow's balance is decreased by p's deposit. This requirement does not hold if we interpret it directly over executions with callbacks. As illustrated in Fig. 3, there could be multiple calls to the escrow before `claimRefund(p)` terminates, which may increase the escrow's balance (via calls to `deposit(p)`) as well as decrease it (via calls to `claimRefund(p)`). One approach to address this discrepancy is to directly account for all possible callbacks in the formal requirement. Unfortunately, this is complicated and error-prone because external contracts can behave arbitrarily: one would need to accommodate infinitely many additional behaviors. Conceptually, this challenge is analogous to specifying concurrent objects [38], [58].

Effectively external callback free contracts. To address the specification challenge, we mirror a well-established approach to specifying concurrent objects [41]. The idea is to let user requirements implicitly assume no callbacks from external contracts. We then lift those requirements to all possible behaviors in a generic way. We do that by imposing an extra condition that we call *effective external callback freedom (EECF)*, which generalizes [38] to our setting. This extra condition stipulates that any behavior with external callbacks is equivalent to a behavior without. Any two equivalent behaviors are guaranteed to execute the same functions and terminate in an identical state. For example, the behavior in Fig. 3 is equivalent to executing `claimRefund(p)` and then `claimRefund(q)` sequentially (without external callbacks).

The EECF condition is satisfied by most real-world contracts. In practice, contracts defer calls to external contracts to the end of transactions, as a generic defense against unexpected state changes due to malicious callbacks in the middle of transactions. Indeed, this best practice provides a sufficient condition to establish EECF. A contract that did not satisfy EECF is the infamous DAO contract [1], which has a function similar to `claimRefund` with the exception that lines 25 and 26 are switched, meaning that p's deposit is set to zero only after the call is made. Indeed, this enables an adversary to drain the escrow's balance (by reclaiming her deposit multiple times).

Property / System	Mythril	Oyente	Manticore	VERX
Inductive, holds: $\varphi_{\mathbf{R1}}$	FP	FP	TN	TN
Violation: $\varphi_{\mathbf{R2}}$	FN	TP	FN	TP

Fig. 4: Running existing symbolic analyzers for verifying $\varphi_{\mathbf{R1}}$ and finding the violation of $\varphi_{\mathbf{R2}}$.

Formalizing requirements. We present the formalization of the requirements for our example in Fig. 2c. The properties are interpreted over a sequence of blockchain states that appear *in-between* transactions, i.e., states that users observe before and after transactions. We deliberately do not define the requirements' semantics over states that appear *within* transactions, as such states capture low-level implementation details and transient computations which are not observable by end-users. As mentioned, the EECF condition *lifts* the requirements to behaviors with callbacks: for any behavior, we impose the given requirements on the equivalent behaviors where the contract's functions are invoked without callbacks.

Property $\varphi_{\mathbf{R0}}$ formalizes that for any state (captured with the *always* temporal quantifier, denoted by $\Box$) reached after a successful execution of `claimRefund(address p)`, the escrow's balance (returned by `Escrow.balance`) equals the previous escrow's balance (denoted `Escrow.balance`$^\bullet$) minus the deposit of investor p in the previous state. The property refers to the p's deposit in the previous state, as it is set to zero after executing `claimRefund`.

Property $\varphi_{\mathbf{R1}}$ formalizes that if the escrow is not in state `SUCCESS` (indicating a successful crowdsale), then the sum of deposits (returned by $sum($`deposits`$)$) is less than or equal to the escrow's balance (returned by `Escrow.balance`).

Properties $\varphi_{\mathbf{R2}}$ and $\varphi_{\mathbf{R3}}$ use the *once* quantifier (denoted by $\blacklozenge$) to refer to past states, where $\blacklozenge\varphi$ holds if φ holds at any past state of the contract. For example, $\varphi_{\mathbf{R2}}$ formalizes that at no point in time the functions `withdraw()` and `claimRefund()` have both been successfully executed by the contract. Property $\varphi_{\mathbf{R3}}$ uses the idiom $\Box(\varphi \rightarrow \neg\blacklozenge\psi)$ to formalize that once ψ holds at a given state, then φ must never hold in the future of the contract.

C. Challenge 2: Unbounded Verification

Next, we discuss the challenges in verifying the properties of our example.

Precise symbolic execution for Ethereum. We start with $\varphi_{\mathbf{R1}}$, a global invariant of the form $\Box\varphi$ with no nested temporal quantifiers. This property is inductive: it holds in the initial blockchain state, and it is also preserved after processing any transaction sent to the contracts at any blockchain state where $\varphi_{\mathbf{R1}}$ holds. To establish this, one can encode all behaviors of the functions using symbolic execution, which is possible due to the limited use of loops/recursion in most contracts, as we discuss in Section VII. However, modeling Ethereum's Virtual Machine (EVM) both *precisely* and *efficiently* has proved challenging. In fact, in Fig. 4, we show that 2 out of the 3 state-of-the-art symbolic execution engines for Ethereum fail

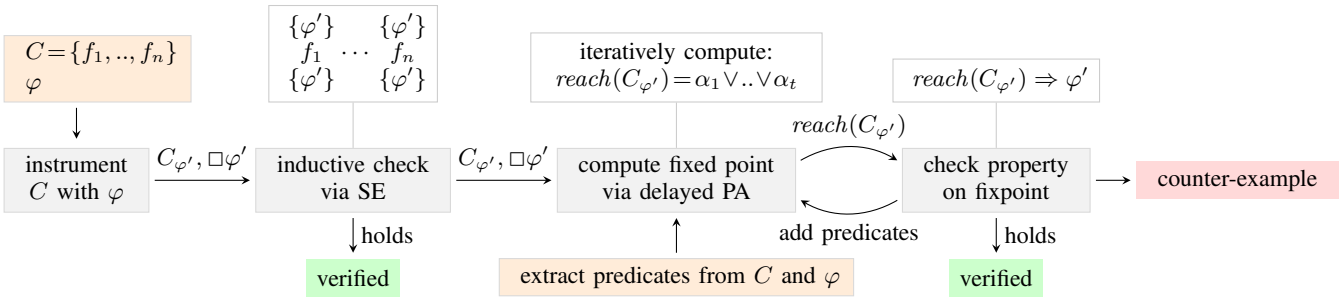

Fig. 5: Flow of VERX: given a contract C and property φ, VERX instruments C with φ and attempts an inductive check using symbolic execution. If this fails, it performs delayed abstraction using predicates extracted from C and φ.

to establish our invariant (i.e., to verify property $\varphi_{\mathbf{R1}}$); we note that Manticore [2] is precise, but inefficient for some examples, as we show in our evaluation (Section IX). We address this challenge by designing a new symbolic execution engine for EVM (Section VIII-A) that is both precise and efficient for a practical fragment of Solidity (Section VII).

Discovering deeper violations. The following sequence of transactions violates property $\varphi_{\mathbf{R2}}$:

1) call to `close()` at state **`now`** $>$ `closeTime` and `raised` $<$ `goal`, which changes the escrow's state to `REFUND`;
2) call to `claimRefund(p)` with any address `p`;
3) call to `invest()` with ether value `msg.value` $\geq$ `goal`;
4) call to `close()`, now at state **`now`** $>$ `closeTime` and `raised` $\geq$ `goal`, which changes the escrow's state to `SUCCESS`;
5) call to `withdraw()`.

This violation is due to a missing pre-condition in function `invest()` (Line 15 in Fig. 2b) which would prevent calls after the crowdsale has ended. This shows that counter-examples in stateful contracts can be deep, going beyond the reach of existing symbolic execution tools (with their default depth limit of 2). In fact, in Fig. 4, we see that both Mythril and Manticore (which unroll multiple transactions) fail to discover the violation with a 5 hours timeout limit and a depth limit set to 5. Oyente discovers the bug due to imprecise modeling of EVM and not due to supporting precise exploration up to depth 5 (as shown in the false positive for $\varphi_{\mathbf{R1}}$). VERX correctly fails to verify this property and outputs exactly the sequence of transactions listed above as a possible violation. As we show in our evaluation, once the missing pre-condition is added, VERX successfully verifies $\varphi_{\mathbf{R2}}$.

Inferring invariants using abstraction. Property $\varphi_{\mathbf{R3}}$ is non-inductive. To verify it, VERX needs to infer an invariant that holds over all reachable contract states. In this example, the invariant is constructed out of the following atomic predicates: **`now`** $>$ `closeTime`, `raised` $<$ `goal`, and $sum($`deposits`$) \leq$ `raised`, where the first two predicates come from function pre-conditions. VERX extracts these and other predicates from the code and the property. To compute the needed invariant, VERX uses delayed abstraction (explained shortly) where the abstraction step is applied only at transaction boundaries. This delay is needed to verify $\varphi_{\mathbf{R3}}$ because the predicate $sum($`deposits`$) \leq$ `raised` is violated between Line 16 and Line 17 in the crowdsale contract. That is, standard predicate abstraction would fail to verify $\varphi_{\mathbf{R3}}$ with these predicates, while delayed predicate abstraction succeeds. Delayed predicate abstraction is explained in Section VI.

III. VERIFICATION FLOW OF VERX

We now describe the verification flow of VERX (illustrated in Fig. 5) and discuss how it addresses key technical challenges. The input to VERX consists of one or more contracts (we call these a bundle of contracts) together with a (temporal) safety property φ. For illustration purposes, we assume we are given one contract C with functions $\{f_1, \ldots, f_n\}$.

Verification by reduction. First, to verify a temporal property of interest φ, VERX processes C and φ so to produce an instrumented contract $C_{\varphi'}$ and a new property $\Box\varphi'$, where $\Box$ is the *always* quantifier and φ' is an assertion over the global blockchain state. For example, it converts $\varphi_{\mathbf{R3}} \equiv \Box($`claimRefund()` $\rightarrow \neg\blacklozenge(sum($`deposits`$) \geq$ `goal`$))$ into $\Box($`claimRefund()` $\rightarrow \neg p_\psi)$, where p_ψ tracks whether $sum($`deposits`$) \geq$ `goal` has been satisfied in the current or the past states of the contract. This approach of reducing temporal property verification to a reachability check was successfully used in prior work (e.g., [26], [28]). A key benefit is that it enables VERX to leverage existing analysis techniques, such as symbolic execution and predicate abstraction (described next) to verify that $\Box\varphi'$ holds on the instrumented contract $C_{\varphi'}$ (and as a result that φ holds on the original contract C).

Verification of inductive properties without abstraction. As stated, the contract's functions are run in an infinite loop:

```
1  while true
2    (user, func, args) := // arbitrary
3    run func(args) as user
```

Each iteration corresponds to the execution of a single transaction, constructed by any user who decides to run an arbitrary contract function with arbitrary arguments. Since this loop is infinite and the number of transactions that users may initiate is unknown in advance, we cannot directly apply symbolic execution (SE) to verify that φ' holds on the instrumented contract (as SE requires loops to be bounded). That is why

we first attempt to verify φ' inductively: assuming φ' holds before f_i is invoked (pre-condition) with symbolic arguments and a symbolic user, we check whether φ' holds after f_i has completed (post-condition). If the inductive check holds for every function f_i and if φ' holds on the initial state then we conclude that $\Box\varphi'$ holds on $C_{\varphi'}$. If the above check fails, however, φ' may still hold for $C_{\varphi'}$. Conceptually, this means we need to strengthen φ' while ensuring that it over-approximates the reachable states of $C_{\varphi'}$.

Verification with delayed predicate abstraction. To over-approximate the reachable states of $C_{\varphi'}$, VERX uses a combination of predicate abstraction and symbolic execution to perform *delayed predicate abstraction*. The idea is to analyze each transaction fully with symbolic execution and abstract only the states where φ' is meant to hold, namely, those appearing at transaction boundaries. The abstraction is done using the predicates automatically extracted from the contract C and the original property φ.

We iterate this procedure in a classic fixed point iteration loop. First, we initialize the set $reach(C_{\varphi'})$ of abstract states reachable at transaction boundaries with the abstraction of the concrete initial state. Then, we simply follow the outer while loop and non-deterministically select a function f_i to invoke; we execute f_i symbolically (assuming symbolic arguments and a symbolic user) and compute the precise symbolic state s at the end of the transaction. Technically, s is a disjunction of constraints $p_1 \vee \cdots \vee p_k$, each p_i capturing a possible path of the function's k paths. After computing s, VERX abstracts each $p_i \in s$ into an abstract state α_i. This results in a set of new abstract states $\alpha_1, \ldots, \alpha_m$ which are added to $reach(C_{\varphi'})$. Note that $m \leq k$, as the symbolic states of two paths may become abstracted to the same abstract state.

Despite the infinite outer loop, VERX is guaranteed to reach a fixed point since the predicate abstraction domain is finite (due to the finite number of abstract predicates). The over-approximation of the reachable states of $C_{\varphi'}$ is given by the disjunction $\alpha_1 \vee \cdots \vee \alpha_t$ of all abstract states in the fixed point $reach(C_{\varphi'})$. To verify the property, VERX checks whether the fixed point logically implies the desired property φ'. If yes, we conclude that $\Box\varphi'$ holds on $C_{\varphi'}$. Otherwise, VERX attempts to construct a counter-example. In doing so, it either succeeds and outputs the counter-example or fails and asks the user to provide additional predicates. We note that one could implement standard abstraction refinement techniques to automate predicate discovery [25].

IV. SYSTEM MODEL

In this section we provide background on the Ethereum blockchain and formalize its behavior. We then formalize the notion of bundle behaviors VERX reasons about.

A. Ethereum Overview

The Ethereum blockchain is a distributed storage that supports two types of accounts: user accounts and contract accounts. User accounts are the entry-points through which users interact with the system by submitting transactions. Contract accounts (or simply contracts) are autonomous objects that process transactions. Every contract is associated with a program that executes incoming transactions, and a private storage for persisting data across transactions.

$$
\begin{aligned}
\langle \mathsf{Execution} \rangle &::= \langle \mathsf{External\ txn} \rangle^* \\
\langle \mathsf{External\ txn} \rangle &::= \langle \mathsf{Internal\ txn} \rangle \\
\langle \mathsf{Internal\ txn} \rangle &::= \langle \mathsf{Message} \rangle [\langle \mathsf{Command} \rangle^*] \\
\langle \mathsf{Message} \rangle &::= (\mathsf{Recipient}, \mathsf{Sender}, \mathsf{Value}, \mathsf{Data}, \mathsf{Gas}) \\
\langle \mathsf{Command} \rangle &::= \mathsf{Load} \mid \mathsf{Store} \mid \mathsf{Local} \mid \langle \mathsf{Internal\ txn} \rangle
\end{aligned}
$$

Fig. 6: An *execution* is a sequence of *external transactions* each nesting one or more *internal transactions*. Each internal transaction starts with a *message* and proceeds in a sequence of *commands*. Commands may load or store data from and to the private storage, perform local computations (not affecting the storage), and initiate nested internal transactions.

$$
\mathsf{Behavior} \triangleq \mathsf{MState}^* \quad \mathsf{MState} \triangleq \mathsf{BState} \times \mathsf{Frame}^*
$$

$$
\mathsf{BState} \triangleq \mathsf{Address} \to \mathsf{Program} \times \mathsf{Storage} \times \mathsf{Value}
$$

$$
\mathsf{Frame} \triangleq \mathsf{Message} \times \mathsf{Program} \times \mathsf{Memory} \times \mathsf{Gas}
$$

$$
\begin{array}{llll}
\mathsf{Program} \triangleq \mathsf{Code} & \mathsf{Recipient} & \triangleq \mathsf{Address} & \mathsf{Data} \triangleq \mathsf{Word}^* \\
\mathsf{Storage} \triangleq \mathsf{Byte}^{\mathsf{Word}} & \mathsf{Sender} & \triangleq \mathsf{Address} & \mathsf{Gas} \triangleq \mathsf{Word} \\
\mathsf{Memory} \triangleq \mathsf{Byte}^{\mathsf{Word}} & \mathsf{Value} & \triangleq \mathsf{Word} & \mathsf{Address} \triangleq \mathsf{Word}
\end{array}
$$

Fig. 7: An EVM *behavior* is a sequence of machine states, represented as pairs holding a blockchain state and a stack of frames, one per nested internal transaction.

An execution of the Ethereum blockchain is a sequence of external transactions, as defined in Fig. 6. Each transaction is initiated with a *message* that defines the addresses of the *recipient* and *sender* accounts, a *value* of ether (possibly zero) to be transferred from the sender to the recipient account, (possibly empty) *data*, and a *gas* value. If the recipient is a user account then the data is empty and the transaction results in transferring the ether value from the sender to the recipient. Otherwise, the recipient is a contract, and the data identifies a function of the contract's program together with arguments passed to the function. Upon the reception of such a message, the contract's program is executed by the Ethereum Virtual Machine (EVM) [66], modifying the contract's storage accordingly. The execution of each command, such as a storage write, is associated with a gas fee and the execution aborts if the accumulated gas fees exceed the gas value.

A contract's program can create messages to start transactions on other contracts, which in turn can do the same. These transactions behave like a standard procedure call, meaning the caller waits for the callee to complete. The calls have transactional semantics and are termed *internal transactions*. Since procedure calls nest, this leads to a nested transactional model: aborting the current internal transaction undoes its

effects on the storage, including the effects of all nested (child) internal transactions. The top-level transaction initiated by the user together with all nested transactions form an *external transaction*. Finally, an execution of the Ethereum blockchain is a sequence of such external transactions.

To illustrate this model, consider that account `User` calls function `close()` of the `Crowdsale` contract in Fig. 2, which in turn calls function `close()` of the `Escrow` contract. The corresponding external transaction is given by

$$
\begin{array}{l}
(\texttt{Crowdsale}, \texttt{User}, 0, \texttt{close()}, 200000)[\\
\quad \cdots \\
\quad (\texttt{Escrow}, \texttt{Crowdsale}, 0, \texttt{close()}, 129275)[\,\cdots] \\
\quad \cdots \\
]
\end{array}
$$

Here, we omit all non-calls (indicated by $\cdots$). This transaction consists of two messages. The amount of ether in both messages is set to 0, indicating that no ether is transferred when processing the transaction. The value $200\,000$ in the first message is the amount of gas provided by `User` for processing the transaction. The gas in the second message ($129\,275$) is lower as reaching the call to the escrow costs $70\,725$ gas.

B. Ethereum Behaviors

A *behavior* of the EVM is the sequence of machine states the EVM goes through during execution. The EVM state of the EVM is formalized in [66], of which we provide a schematic description in Fig. 7. A *machine state* is a pair holding the *blockchain state*, and a stack of *frames*, one per active nested internal transaction.

- The blockchain state collects all account data and is a mapping from account addresses to account states. The figure shows only contract accounts, whose state consists of the contract's program, storage, and balance (values).
- A frame contains: (1) the message used to activate the call, (2) the next program fragment to be executed (i.e., a program counter), (3) the call's local scratchpad *memory*, (4) the remaining gas available for execution.

The EVM operates on 32-byte words. All numeric values (e.g. addresses) are represented as unsigned integer words. Contract storage and frame memory are word-addressed.

C. Bundle Behaviors

The requirements of a bundle $\mathcal{C}$ of contracts constrain its EVM behaviors. Given an EVM behavior, we obtain the behavior of the bundle by extracting the sequence of pairs (T, b) where T is a transaction whose recipient is in the bundle $\mathcal{C}$ and b is a blockchain state that captures the effects of T. As discussed in Section II-B, we follow the well-established approach in the field of concurrency and extract behaviors of $\mathcal{C}$ only from EVM behaviors without external callbacks (i.e., without concurrency). Thus, our specifications do *not* directly constrain all possible EVM behaviors but only those where transactions run isolated from external callbacks. The remaining EVM behaviors are controlled by imposing a generic correctness criterion [38] on the bundle $\mathcal{C}$.

Definition 1. *A* $\mathcal{C}$-entry transaction *in an EVM behavior is a transaction executed by a contract in* $\mathcal{C}$ *whose parent transaction, if present, is executed by a contract not in* $\mathcal{C}$*.*

For example, in the EVM behavior in Fig. 3 the transactions `Escrow.claimRefund(p)` and `Escrow.claimRefund(q)` are $\mathcal{C}$-entry transactions: the first one is an external transaction that executes contract `Escrow` and does not have a parent transaction; the second one executes contract `Escrow` and its parent transaction is executed by a contract $p \notin \mathcal{C}$ outside the bundle $\mathcal{C} = \{\texttt{Escrow}, \texttt{Crowdsale}\}$.

We call an EVM execution $\mathcal{C}$*-external callback-free* if it contains no $\mathcal{C}$-entry transaction nested in another $\mathcal{C}$-entry transaction. For example, the behavior shown in Fig. 3 is not $\mathcal{C}$-external callback-free because `Escrow.claimRefund(q)` is nested inside `Escrow.claimRefund(p)`.

Let $\mathsf{msg}(T)$ be the message that initiated T, and let $\mathsf{post}(T)$ stand for the blockchain state at the point when T finishes.

Definition 2. *The* bundle behavior of $\mathcal{C}$ *corresponding to a* $\mathcal{C}$*-external callback-free EVM execution is the sequence that records the pair*

$$\big(\mathsf{msg}(T), \mathsf{post}(T)\big)$$

for every $\mathcal{C}$*-entry transaction* T *in the EVM execution.*

That is, for every transaction T, the behavior records the transaction message and the blockchain state at the end of T but restricted only to contracts in the bundle $\mathcal{C}$.

Effective external callback freedom. The criterion that we impose on bundles requires that all EVM behaviors are in a certain sense equivalent to $\mathcal{C}$-external callback-free executions. We adapt the *effective callback freedom* criterion of Grossman et al. [38] to our setting where the internal callbacks within a $\mathcal{C}$-entry transaction are thought of as executing sequentially. Both, our criteria and [38], are variants of conflict-serializability [52]. For brevity, we do not state the actual criterion but a necessary and sufficient condition. As standard, we formalize the condition by first constructing a directed graph (digraph) that captures dependencies between operations and then requiring that the resulting digraph is acyclic.

Definition 3. *The* $\mathcal{C}$-serialization digraph *of an EVM execution has a vertex for every* $\mathcal{C}$*-entry transaction and an edge* (S, T) *for all operations* s *of* S *and* t *of* T $(S \neq T)$ *such that:*

1) s *precedes* t *in the EVM execution, and*
2) *at least one of* s *and* t *modifies part of the blockchain state that is read or modified by the other.*

Intuitively, each edge (S, T) indicates that the blockchain state after T depends on S being executed earlier. $\mathcal{C}$-entry transactions not involved in circular dependencies are not logically interrupted by callbacks outside the bundle.

Definition 4. *A bundle* $\mathcal{C}$ *is* effectively external callback free *if all EVM executions have acyclic* $\mathcal{C}$*-serialization digraphs.*

V. Property Specification Language

We express properties of bundle behaviors using past linear temporal logic, or Past LTL for short. We provide a brief introduction below, for further details please refer to [48].

Linear temporal logic. A formula in Past LTL is inductively defined as either an atomic formula A, or a logical or temporal connective applied to one or more formulas φ, ψ:

$$\varphi, \psi ::= A \mid \varphi \vee \psi \mid \neg\varphi \mid \bullet\varphi \mid \varphi \mathcal{S} \psi$$

A formula can be interpreted at any position in a bundle behavior. Atomic formulas directly refer to the message or blockchain at the current position. For example, the formula `withdraw()` holds when the current transaction was a call to the `withdraw()` function. The formula from Fig. 2

$$sum(\texttt{deposits}) \leq \texttt{Escrow.balance}$$

holds when in the current state the sum of values of the `deposits` mapping is less than or equal to the balance of the `Escrow` contract. Boolean connectives are interpreted in the standard way. For the temporal connectives

- *previously* $\bullet\varphi$ evaluates φ in the previous state if such exists and returns false otherwise;
- *since* $\varphi \mathcal{S} \psi$ looks for a nonfuture moment when ψ holds so that φ holds in all moments after, up to the present.

The since connective can define other useful connectives like *once* $\blacklozenge$ (is/was true) and *still* $\blacksquare$ (is/has always been true):

$$\blacklozenge\psi \triangleq \top \mathcal{S} \psi \qquad \blacksquare\psi \triangleq \neg\blacklozenge\neg\psi$$

Given a Past LTL formula φ, the $\square\varphi$ formula uses the always connective $\square$ and states that φ must hold at all positions of the bundle behavior.

Conversion to non-temporal formulas. Past LTL formulas are sufficiently expressive to capture any safety property, yet they are equivalent to assertions, i.e., non-temporal formulas. This allows us to convert them to assertions and reuse standard safety verification techniques.

To convert a Past LTL formula φ to a non-temporal one, we encode the semantics of past connectives into the behaviors of the contract. That is, we introduce auxiliary variables that track the truth values of temporal subformulas in φ, a well-known method [54].

We demonstrate this method with an example. Consider the temporal formula $\varphi_{\mathbf{R3}} = \square\varphi$ from Fig. 2:

$$\varphi \equiv \texttt{claimRefund()} \rightarrow \neg\blacklozenge(sum(\texttt{deposits}) \geq \texttt{goal}).$$

First, we select a subformula whose top-level connective is temporal. Only one such subformula is present in our example:

$$\psi \equiv \blacklozenge(sum(\texttt{deposits}) \geq \texttt{goal}).$$

Then, we introduce a fresh variable p_ψ that tracks ψ's truth value, and update it according to the semantic rule of $\blacklozenge$:

$$p'_\psi \equiv (sum(\texttt{deposits}) \geq \texttt{goal}) \vee p_\psi.$$

Finally, we rewrite φ to the non-temporal formula:

$$\varphi \equiv \texttt{claimRefund()} \rightarrow \neg p_\psi.$$

Property syntax. To allow auditors and developers to specify properties, we extend Solidity's syntax with temporal quantifiers. For example, we denote the operators

- always $\square$ by `always`,
- once $\blacklozenge$ by `once`, and
- previously $\bullet$ by `prev`.

For convenience, we also provide aggregate functions over mappings, denoted by `sum(deposits)`. The properties can further refer to the latest transaction's data, including its signature and arguments. We use Solidity's syntax for logical connectives and expressions (e.g., users write `&&` for $\wedge$) and introduce implication (`==>`) which is often used in properties.

VI. Delayed Abstraction for Verification

To verify a Past LTL specification of a bundle of contracts $\mathcal{C}$ we apply abstract interpretation over a symbolic domain. We employ predicate abstraction [35] but without the usual conversion to boolean programs. Our approach is similar to that of Flanagan and Qadeer [32] where two transformers are alternated: precise symbolic transformers to handle individual commands, and an imprecise transformer to ensure convergence. In contrast, classic abstraction applies an imprecise transformer at every step. Hence, we call the precise/imprecise approach *delayed abstraction*.

VeriX receives a bundle $\mathcal{C}$ of contracts (written in Solidity) and a Past LTL specification $\square\varphi$ that should hold for all behaviors of the bundle. First, it converts the Past LTL specification to an assertion (Section V). Thus, at this point, we assume that φ is an assertion. Then, VeriX compiles all contracts in $\mathcal{C}$ to EVM bytecode [66], and then it performs abstract interpretation on the bytecode. We compile to EVM because it is much simpler than Solidity.

To keep the discussion concise, we assume the bundle $\mathcal{C}$ is described by the blockchain address of its contracts together with two programs:

1) $\mathsf{init}_\mathcal{C}$—a deterministic program that produces the initial blockchain state in every behavior of the bundle;
2) $\mathsf{step}_\mathcal{C}$—a program that non-deterministically executes a transaction of $\mathcal{C}$ starting in a given bundle state.

Given the property φ and the compiled code of $\mathcal{C}$, VeriX tries to verify the following program:

```
initC; assert φ; while true { stepC; assert φ }
```

Ignoring the assertions, this program produces all possible behaviors of the given bundle of contracts $\mathcal{C}$.

This verification problem amounts to proving that φ holds for all message-state pairs (m, s) appearing in behaviors of $\mathcal{C}$. Recall that the behaviors of a bundle were defined in Section IV. This set of pairs is called the *concrete semantics* of $\mathcal{C}$, which we denote with $\mathsf{Sem}(\mathcal{C})$. Thus, our goal is to verify:

$$\mathsf{Sem}(\mathcal{C}) \models \varphi. \tag{1}$$

A. Abstract Interpretation

As the concrete semantics can be infeasible to compute, establishing (1) directly is not possible except for restricted cases. Abstract interpretation [27] addresses this issue by soundly approximating the concrete semantics with an *abstract semantics* $\mathsf{Sem}^{\#}(\mathcal{C})$ for which the verification problem is easier:

$$\mathsf{Sem}^{\#}(\mathcal{C}) \models \varphi. \tag{2}$$

Sound approximation means that the set $\gamma(\mathsf{Sem}^{\#}(\mathcal{C}))$ of all message-state pairs encoded by the abstract semantics extends the concrete semantics:

$$\mathsf{Sem}(\mathcal{C}) \subseteq \gamma(\mathsf{Sem}^{\#}(\mathcal{C})).$$

Then, because of soundness, proving (2) implies that (1) holds.

To compute the abstract semantics by abstract interpretation, one mimics how the concrete semantics is computed as an element of the powerset lattice $\mathbb{C} = \mathcal{P}(\mathbb{S})$, where $\mathbb{S}$ is the set of all possible message-state pairs, including those that do not appear in the behaviors of the bundle $\mathcal{C}$. We reinterpret the two programs that describe the bundle as *concrete transformers*:

$$\mathsf{init}_{\mathcal{C}} \in \mathbb{C} \qquad\qquad \mathsf{step}_{\mathcal{C}} : \mathbb{C} \to \mathbb{C}.$$

The concrete semantics $\mathsf{Sem}(\mathcal{C})$ is simply the least fixed point of the monotone map

$$S \mapsto \mathsf{init}_{\mathcal{C}} \cup \mathsf{step}_{\mathcal{C}}(S).$$

To compute $\mathsf{Sem}^{\#}(\mathcal{C})$ by abstract interpretation, we first replace the *concrete domain* $\mathbb{C}$ with an *abstract domain* $\mathbb{A}$, where the two are connected by the *concretization function*

$$\gamma : \mathbb{A} \to \mathbb{C}.$$

We then define *abstract transformers* instead of concrete ones:

$$\mathsf{init}^{\#}_{\mathcal{C}} \in \mathbb{A} \qquad\qquad \mathsf{step}^{\#}_{\mathcal{C}} : \mathbb{A} \to \mathbb{A}.$$

To mimic the concrete computation, we equip the abstract domain $\mathbb{A}$ with a lattice structure, that is, a join operation $\sqcup$. This allows us to define the abstract semantics $\mathsf{Sem}^{\#}(\mathcal{C})$ as a fixed point of the map

$$A \mapsto \mathsf{init}^{\#}_{\mathcal{C}} \sqcup \mathsf{step}^{\#}_{\mathcal{C}}(A).$$

As a result, we obtain a sound approximation provided the abstract transformers are *sound*. That is, for all $A \in \mathbb{A}$:

$$\mathsf{init}_{\mathcal{C}} \subseteq \gamma(\mathsf{init}^{\#}_{\mathcal{C}}) \qquad \mathsf{step}_{\mathcal{C}}(\gamma(A)) \subseteq \gamma(\mathsf{step}^{\#}_{\mathcal{C}}(A)).$$

Fig. 8 summarizes the abstract interpretation approach. To model non-determinism more precisely, we use a powerset domain $\mathbb{A}$, that is, the elements $A \in \mathbb{A}$ are sets of elements of another domain. The concretization function then is:

$$\gamma(A) = \bigcup_{a \in A} \gamma(a).$$

```
1:  Input
2:      C    A bundle of contracts.
3:      φ    A non-temporal formula.
4:  procedure VERIFY(C, φ)
5:      A ← FIXPOINT(A ↦ init#_C ⊔ step#_C(A))
6:      for all a ∈ A do
7:          if a ⊭ φ then
8:              report a and exit.
9:          end if
10:     end for
11:     report φ holds.
12: end procedure
```

Fig. 8: Verification via a powerset domain.

B. Delayed Abstraction

We now discuss the particular abstraction used in VERX. We apply what we call a *delayed abstraction*, an elegant idea which first appeared in [32]. Delayed abstraction addresses imprecision in reasoning about invariants that are broken at intermediate points of the program. For smart contracts this typically occurs when a transaction temporarily breaks its invariant inside and then restores it at the end of its execution.

To deal with imprecision, delayed abstraction simply delays the abstraction step until the end of the transaction, but employs a precise symbolic domain Φ to reason within the transaction. Because fixed point computation in the precise domain might not converge, at the end of the transaction, a switch is made to a less precise predicate abstraction domain $\mathbb{A} = \mathcal{P}(\mathbb{P}\mathbb{A})$ where $\mathbb{P}\mathbb{A} \subseteq \Phi$. Thus, the abstract transformer first makes a precise symbolic execution step, and then a predicate abstraction step:

$$\mathbb{A} \xrightarrow{\mathsf{SE}(f)} \mathcal{P}(\Phi) \xrightarrow{\alpha} \mathbb{A}.$$

To perform symbolic execution, the symbolic domain Φ consists of first order formulas with message-state pairs (m, s) for models. That is, for every $\psi \in \Phi$

$$\gamma(\psi) = \{(m, s) : (m, s) \models \psi\}.$$

The only restriction to the formulas in Φ is that an SMT solver should be able to handle them. The $\mathsf{SE}(f)$ step starts with a set of formulas $A \in \mathbb{A}$ and for each formula $\psi \in A$, it symbolically executes all feasible program paths in $f \in \{\mathsf{init}_{\mathcal{C}}, \mathsf{step}_{\mathcal{C}}\}$. The result of processing ψ is a set of formulas $\{\psi'_1, \ldots, \psi'_n\}$ (one per feasible program path) that give the strongest postcondition of f with respect to ψ:

$$\gamma(\bigvee_{i=1}^{n} \psi'_i) = f(\gamma(\psi)).$$

The result of processing all formulas in A, as described above, is an intermediate set $S \in \mathcal{P}(\Phi)$ of formulas.

For delayed abstraction, we use a set $\mathbb{P}\mathbb{A}$ of *cubes* [35] which are conjunctions of literals over a fixed set of basic predicates $P = \{P_1, \ldots, P_n\} \subseteq \Phi$. The abstraction step

$$\alpha : \Phi \to \mathbb{P}\mathbb{A}$$

projects each formula $\psi' \in S$ onto $\mathbb{PA}$:

$$\alpha(\psi') = \bigwedge\{L \in \Lambda : \psi' \models L\},$$

where Λ is the set of all literals over the set P. The result of processing the set S results in a set of cubes that is an element in the domain $\mathbb{A}$.

An important part of predicate abstraction is choosing the set P of basic predicates. If one does not find the right set P, the computed abstract element will be sound, but might be too imprecise to prove the desired specification φ. Typically, one seeds P with all atomic subformulas of the specification. If verification fails with this set, we increase P automatically by extracting predicates that appear in the program. We define the types of predicates we extract from the program and give examples where they bring sufficient precision to prove realistic properties in Section IX-C. If verification still fails, we ask the user to provide additional predicates (in principle, one could extend the system to perform standard abstraction refinement so to automatically discover predicates).

VII. Practical Solidity Fragment

The effectiveness of delayed abstraction relies on a precise symbolic execution engine that scales to realistic contracts. Building such an engine for arbitrary Solidity contracts is, however, challenging due to the Turing-completeness of the language. In this section, we define a practical fragment of Solidity that covers a wide range of real-world contracts yet enables precise and scalable symbolic execution. We summarize the restrictions in Fig. 9 and explain them in detail below.

A. Loops and Recursion

Symbolic execution fails to terminate when a program contains unbounded loops or recursion. In Ethereum, however, long transaction executions are costly (since they consume more gas) and moreover cannot exceed the block's gas limit, which imposes a hard bound on the gas budget of transactions. Because of this, unbounded loops and recursion are considered anti-patterns.

In our fragment, we consider Solidity contracts without recursion and where loops are bounded. Such contracts have finitely many paths which can be fully analyzed symbolically. To further extend our fragment, we also support the following common loop pattern in Solidity:

```
1  function pm(Arg[] args) {
2    for (i = 0; i < args.length; i++) {
3      f(args[i]);
4    }
5  }
```

We call this loop pattern *packed-calls*, as the loop is merely used to group multiple independent calls to the function `f` in a single transaction. For the purpose of verification, such loops can be eliminated without changing the semantics of the bundle. That is, any transaction invoked with multiple elements in `args` can be split into multiple transactions to `pm`:

$$\texttt{pm}([a_1, \ldots, a_n]) \simeq \texttt{pm}([a_1]) \ldots \texttt{pm}([a_n]). \quad (3)$$

VERX automatically detects and soundly bounds such loops.

Type	Restriction
Loops	Loops are bounded or match the packed-calls pattern
Recursion	No recursion allowed
External calls	Must satisfy effective external callback freedom
Gas exceptions	Gas exceptions propagate to the external transaction
Storage access	No direct access to storage via assembly
Restricted behaviors	No execution of external code No creation or destruction of contracts

Fig. 9: Solidity fragment supported by VERX

B. Calls to External Contracts

The Ethereum blockchain permits a bundle contract c_1 to call an *external contract* d, and then contract d to call back some bundle contract c_2:

$$c_1 \in \mathcal{C} \rightarrow d \notin \mathcal{C} \rightarrow c_2 \in \mathcal{C}. \quad (4)$$

The external contract d can potentially make an unbounded number of callbacks to the bundle (until all gas is depleted), making it infeasible to fully analyze with symbolic execution.

To address this challenge, we consider only contracts that satisfy the effective external callback freedom condition defined in Section IV-C. This condition implies that any behavior of the contracts with external callbacks is equivalent to a behavior without external callbacks. We defined the check used to determine whether a smart contract satisfies this condition in Section VIII.

C. Gas Exception Propagation

If an internal transaction consumes all gas allocated to it, then it terminates with an "out of gas" exception, undoing all changes to the shared storage. After that, execution resumes in the parent transaction. Since the amount of gas can be arbitrary, the above behavior implies that most calls are a potential branching point where the call can be omitted. These branching points explode the number of program paths to be symbolically executed.

We avoid this explosion by requiring contracts to propagate all exceptions up the call stack, effectively reverting the current bundle transaction. This means that the transaction will not appear in the blockchain execution and can be ignored.

D. Storage Access

As mentioned in Section VI, we do not analyze Solidity directly but the EVM bytecode obtained by compiling Solidity code. The challenge with EVM bytecode is that it is low-level and important information is lost during compilation. The major challenge comes when reasoning about compound objects such as structures, arrays, and maps. The EVM lacks primitives for such objects, and consequently, all arrays and maps are spliced together into a single flat array of 2^{256} words. Therefore, to reason about compound objects, we need to reason about the specific hash-based allocation scheme used by the Solidity compiler.

To illustrate the scheme, consider an array a that holds objects of size n. The array gets a unique identifier $\mathsf{id}(a)$, and its elements are distributed according to[1]:

$$a[i] \mapsto \texttt{SHA-3}(\mathsf{id}(a)) + n * i.$$

That is, the array is allocated at the address that equals the SHA-3 hash of its identifier and spans the subsequent words.

The allocation scheme must be *valid* in the sense that distinct objects are allocated disjoint storage. This validity rests on two conditions, which are required to model in order to analyze array and map accesses precisely:

1) There are no SHA-3 collisions to guarantee that distinct objects and map elements start at distinct addresses.
2) SHA-3 hashes are sufficiently spread apart so that distinct objects do not overlap.

These two conditions are assumed to hold for any Solidity contract. They can, however, be violated if a contract directly accesses storage using inlined assembly. To this end, we restrict the use of inlined assembly in bundle contracts.

E. Unsupported Instructions

Finally, in our fragment we ignore contracts that:

1) Execute external code in the context of the bundle contracts (instructions `CALLCODE` or `DELEGATECALL`).
2) Destroy bundle contracts or create new ones (instructions `CREATE` and `SUICIDE`).

Execution of external code is known to entail a security risk. For example, attackers can execute arbitrary code if they gain control over the destination address of a `DELEGATECALL` instruction. Due to these security risks, these instructions are considered a bad practice and are excluded from new bytecode proposals that aim to improve the security of the EVM [44]. Creation and destruction of contracts can be supported but is less common and we omit these behaviors for simplicity.

VIII. The VerX System

We now present details on our VerX system: the new symbolic execution engine, the types of predicates extracted from contracts, and optimizations used to scale verification.

A. Symbolic Execution Engine

VerX uses a new symbolic execution engine for EVM, which avoids the pitfalls of existing engines, as detailed in our evaluation (Section IX-D). Our symbolic execution engine extends standard techniques [19], [45] with the following unique features of Solidity: (i) object allocation in Solidity (determined by hash functions), (ii) contract calls, and (iii) sound approximation of gas mechanics. We also model sums of arrays and mappings, which although not part of Solidity are needed for verifying relevant properties.

Symbolic state. We perform symbolic execution with Z3 [31]. Machine words in the symbolic state are encoded precisely as 256-bit vectors. EVM storage and memory are encoded in the theory of arrays. Calls to external contracts are modeled with uninterpreted constants.

Checking effective external callback freedom (EECF). We check that all contracts in the bundle satisfy EECF through a proxy pattern. The pattern forbids:

1) Writing to the bundle's blockchain state after calls to external contracts. Example writes include transferring ether or writing values to storage variables;
2) Reading the bundle's blockchain state after calls to external contracts with ≥ 5000 gas. Example reads include accessing a contract's balance or storage variables.

The latter condition allows reads for calls to external contracts with < 5000 gas since such calls guarantee that the callee cannot, due to the limited gas budget, modify the bundle's blockchain state. We note that such calls are common because Solidty's statements `send` and `transfer` are frequently used by developers to transfer ether to other contracts and these provide 2300 gas to the callee.

Checking exception propagation. Standard function invocations in Solidity automatically propagate exceptions. Only low-level calls such as `send` and `call` do not propagate them. To check that the caller propagates exceptions, we perform dataflow analysis to verify that the contract always reverts the state when such a call fails.

Modeling object allocation. As mentioned, a unique aspect of Solidity/EVM is the hash-based allocation scheme used to determine the location of an object in the storage by applying the `SHA-3` hash function. The challenge here is that `SHA-3` is a rather complex function and if we model it precisely with SMT constraints, we would cripple the symbolic execution.

To this end, we interpret `SHA-3` as a different, much simpler function, that still ensures the two assumptions mentioned above, namely no hash collisions and sufficiently spread-out hashes. This approach is sound provided that the smart contracts we verify always access memory through Solidity's primitives, that is, they never access raw memory directly. We ensure this memory access by forbidding the use of inline assembly in the Solidity source.

We encode the simpler function with the function symbols

$$\texttt{SHA-3}_n : \{0,1\}^n \to \{0,1\}^{256},$$

one per input size n to a `SHA-3` invocation in the bytecode. We encode the expected behavior of the function with extra SMT constraints. Importantly, the constraints do not specify the function completely. That is, we assume little about the function, and the VerX verification results hold for any function that satisfies those weak assumptions.

(i) The first set of constraints ensures that no collisions occur. No collisions in principle means that the union of all the functions $\texttt{SHA-3}_n$ is one-to-one. This is, of course, impossible since the domain of the function is larger than its codomain. That is why we do not ensure injectivity on its whole domain. Instead, we consider each symbolic execution separately and

[1]For simplicity, we assume that $n = 256$. The storage address is computed using a more complex formula if $n \neq 256$; cf. [8].

ensure injectivity only on the (possibly symbolic) arguments that appear in the execution. To do that, let X_n be the arguments applied to $\texttt{SHA-3}_n$. We add the constraints:

1) No collisions within each X_n:

$$\bigwedge_{x \in X_n} \texttt{SHA-3}_n^{-1}(\texttt{SHA-3}_n(x)) = x. \tag{5}$$

Here, $\texttt{SHA-3}_n^{-1}$ is a function symbol that represents a pseudo-inverse function of $\texttt{SHA-3}_n$.

2) No collisions between each X_m and X_n, for sufficiently many pairs $m \neq n$ of distinct input sizes:

$$\bigwedge_{x \in X_n} L_n \leq \texttt{SHA-3}_n(x) \leq R_n. \tag{6}$$

The numbers $L_i \leq R_i$ divide the address space into disjoint intervals $[L_m, R_m] \cap [L_n, R_n] = \varnothing$.

In our experiments, we set concrete bounds L_i, R_i that are far apart from each other, giving intervals of length 2^{100}.

We used the above constraints instead of the prosaic

$$x \neq y \implies \texttt{SHA-3}_m(x) \neq \texttt{SHA-3}_n(y).$$

because they are linearly many in the size of the union $\cup_i X_i$ instead of quadratic. In our experience, this reduced the load on the SMT solver significantly.

(ii) The second set of constraints ensures that hashes are spread apart. For every set X_n we add the constraints

$$\bigwedge_{x \in X_n} \texttt{SHA-3}_n(x) \equiv 0 \pmod{S}, \tag{7}$$

where S is sufficiently large (64 in our experiments).

Calls. When the engine reaches a call, it checks whether the address being called is concrete or symbolic. If it is concrete, the execution continues by transferring the given (possibly symbolic) value of ether and then checks whether the address is in the bundle. If so, this is a contract account, and we model the call precisely, activating the contract's program in a new frame. If the concrete account is external, then we treat it as a user account, meaning no further action beyond a transfer is taken (as mentioned, this treatment is sound when the bundle contracts satisfy effective external callback freedom).

In the case where the address being called is symbolic, the execution first non-deterministically selects the address of a bundle contract or treats the symbolic address as an external user account. After that, the execution proceeds as before.

Gas mechanics. The engine implements gas mechanics to terminate symbolic execution along infeasible paths, where an out-of-gas exception is guaranteed. Concretely, the engine tracks a lower-bound g_{lower} on the gas cost of each symbolic path. We note that we cannot track the precise gas cost simply because the gas fees for some operations depend on their arguments, and therefore the concrete executions that follow a symbolic path may have different gas costs. The engine also tracks the gas budget g_{budget} specified in the message initiating the call. There are two common cases for the gas budget: (1) it is often unbounded, as it is specified by the user, and (2) it is a constant, for calls initiated by `send` and `transfer` which always provide $2,300$ gas to the callee. Finally, our engine considers any paths where $g_{lower} > g_{budget}$ as infeasible, as they are guaranteed to revert due to an out-of-gas exception.

Tracking sums. Another feature of our engine is to track the sums of arrays and mappings with numeric values. Although these sums are not a feature of the EVM, they enable to specify important properties (e.g., Fig. 2). The sum of an object a is kept in a designated variable $sum(a)$ in the symbolic store. For each write $a' = a[x \mapsto y]$, the sum changes as:

$$sum(a') = sum(a) - a[x] + y.$$

B. Extracting Predicates

Delayed abstraction needs a sufficient set of predicates to successfully verify properties. Based on our experience in verifying contracts (see Section IX), we identify five important kinds of predicates:

1) *Property:* atomic formulas from the property. For example, $\varphi_{\mathbf{R1}}$ (Fig. 2) has two atomic formulas: `state` $\neq$ `SUCCESS` and $sum($`deposits`$) \leq$ `Escrow.balance`.
2) *Points-to:* encode pointers to contract instances. For our motivating example, we extract predicates that capture that `escrow` points to the escrow instance, and its field `owner` points to the crowdsale instance.
3) *Enum/Bool*: encode values of boolean and enumerated fields. For our motivating example, we extract three predicates to encode the value of variable `state`.
4) *Constant*: encode values of constant variables. For our motivating example, we extract `goal` $= 10000 \times 10^{18}$.
5) *Other*: predicates defined by users.

VERX automatically extracts all predicates, except those classified as *Constant* and *Other*, by traversing the contract's abstract syntax tree (AST) to detect contract fields of type boolean, enum, or contract, and defining predicates to capture their values. For a boolean field `b`, VERX adds the predicate `b` $=$ **`True`**. For an enum field `e`, VERX finds its size n and adds the predicates `e` $= 0, \ldots,$ `e` $= n-1$. For example, for variable `state` from Fig. 2(d), VERX adds the predicates `state` $= 0$, `state` $= 1$, and `state` $= 2$. For a contract field `c` of type t, VERX adds the predicate `c` $= t$. This predicate captures the type of `c`, and also serves as a points-to predicate: in our benchmarks, there is one instance per contract type, and thus if the predicate `c` $= t$ holds, then VERX infers the exact instance of `c`. In the future, we plan to extend VERX with points-to analysis to improve precision for more complex cases.

C. Abstraction of non-linear arithmetic

To further scale the analysis, VERX supports abstraction of non-linear arithmetic. Non-linear arithmetic poses a challenge to SMT solvers, yet it is unnecessary to prove many real-world properties. That is why VERX has the option to abstract non-linear computations (e.g., multiplication) by substituting them with uninterpreted functions. The user of the system can disable or enable the abstraction whenever needed, speeding up the analysis for the majority of cases.

IX. Evaluation of VerX

We now present our evaluation of VerX on real-world Ethereum projects. We focus on three key questions:

1) What types of safety properties are common for real-world contracts?
2) How effective is VerX in verifying safety properties of these smart contracts?
3) How precise is the symbolic execution engine of VerX compared to existing symbolic tools for Ethereum?

All of our experiments were run on a server with Intel i7-6700 CPU with 4 cores at 3.4 GHz, 64 GB memory, running Ubuntu 18.04.2 and Python 3.6.

A. Benchmarks and Properties

We now report on our benchmarks and the common types of requirements that we formalized. All our benchmarks are available at https://github.com/eth-sri/verx-benchmarks.

Benchmarks. Our benchmarks consist of the overview example (Fig. 2) and 12 real-world Ethereum projects. Most of these projects are deployed on Ethereum and have over 500 lines of Solidity code. We provide relevant statistics in Table I. The prevalence of crowdsales and tokens in the benchmarks reflects that many of the real-world deployed Ethereum contracts are of these types. We note that some of the verified contracts hold a significant amount of funds ($>$ 100M worth of USD).

For each project, we wrote a deployment contract to define the initial state of the project's contracts. As mentioned, some of these projects have been professionally audited by experts where any property violations have been fixed. Our results use the fixed versions where all properties verify.

Properties. The typical process to elicit properties is to inspect the documentation and to formalize key requirements. To provide an insight into the types of common requirements, we classify them into five distinct categories and show these in Table II, along with concrete examples from our benchmarks:

1) *User-based access control* requirements define which users can invoke critical functions. The example in Table II, taken from the Crowdsale project (part of Zeppelin's projects), stipulates that only the primary (a specific user address) can invoke function `deposit`. We note that attribute-based authorizations, such as "only users with more than 1000 tokens can calls function `foo`" are also supported and common.
2) *State-based properties* define at which states certain authorizations and invariants must hold. The example in Table II, taken from the VUToken project, permits calls to function `withdrawTokens` after the closing time.
3) *State machine requirements* define which state transitions of the contracts are allowed. To track state, contracts often have explicit `state` variables. These properties often involve temporal quantifiers (e.g., **`once`**). The example in Table II, taken from the ICO project, stipulates that once the contract is in state `Refund`, it cannot transition to state `Finalized`, and vice versa.

Project	Type	# Contracts	# Func.	# Ifs	LOC
Overview	Crowdsale	3	7	1	66
Alchemist	Exchange	6	31	2	401
Brickblock	Token	9	36	4	549
Crowdsale	Crowdsale	20	86	2	1,198
ERC20	Token	6	37	1	599
ICO	ICO	14	51	13	650
Mana	Crowdsale	18	66	4	885
Melon	Token	5	35	2	462
MRV	Token	7	49	11	868
PolicyPal	Crowdsale	10	40	10	815
VUToken	Crowdsale	19	70	4	1,120
Zebi	Crowdsale	12	99	9	1,209
Zilliqa	Token	9	31	4	377

TABLE I: Statistics on Ethereum projects.

4) *Invariants over aggregates:* Contracts often store state in mappings. Since Solidity does not allow contracts to iterate over the keys of a mapping, contracts often store state in auxiliary variables to track aggregate information. Invariants over these help establish their correctness. The invariant in Table II, taken from the MRV contract, states that the sum of balances always equals the total supply of tokens, a common token requirement.
5) *Multi-contract invariants:* Finally, contracts often impose invariants over the bundle of contracts in the project. The example in Table II, taken from the MANA project, states that if the continuous sale has not started, the owner of the token is the crowdsale contract.

B. Verifying Contracts using VerX

We now report on the effectiveness of VerX in verifying our benchmarks. We present our results in Table III. VerX successfully verified all 83 properties we considered.

To present our results, we split the properties based on whether they are verified inductively or using abstraction. For each property type, we show the number of verified properties, the average number of symbolic states, and the average execution time in seconds. For properties verified with abstraction, we show the total number of predicates required to verify all properties. For example, the 3 properties of project Brickblock are verified with abstraction, and we report 3 Enum/Bool predicates because each property required 1 boolean predicate.

Our results illustrate several important points. First, 20% of properties can be verified inductively.

Second, the average verification time is 114 seconds for inductive properties and 1296 seconds ($\approx$ 20 minutes) for non-inductive ones. Two outliers are the Mana and PolicyPal projects, which took hours to verify some properties due to complex constraints. To speed up VerX, one can implement standard optimizations, such as slicing program paths that are guaranteed to not change the abstract predicates' truth values.

Third, the number of symbolic states demonstrates the complexity of the analysis. For some properties, VerX explores over a hundred abstract states to verify the property.

Type	Properties	Description	Example
User-based access control	13	Define which users (identified by address or attributes) can invoke certain functions	`always(RefundEscrow.deposit(address)==> (msg.sender == RefundEscrow._primary))`
State-based properties	44	Define invariants and access control that must be enforced in certain states	`always(ICOCrowdsale.withdrawTokens()==> (now > ICOCrowdsale.closingTime))`
State machine requirements	15	Define which state-transitions of the contract are correct	`always(!(once(Escrow.state == SUCCESS) && once(Escrow.state == REFUND))`
Invariant over aggregates	8	Invariants expressed over aggregate values stored in mappings and arrays	`always(MRVToken.totalSupply == sum(MRVToken.balances))`
(Multi)contract invariants	3	Invariants that span across one or more contracts	`always((MANAContinuousSale.started != true)==> MANAToken.owner == MANACrowdsale)`

TABLE II: Common types of properties encountered during audits and concrete examples taken from the benchmarks.

	Inductively Verified			Verified via Delayed Abstraction							
							Total number of predicates (all properties)				
Project	Verified	Avg. states	Avg. time (s)	Verified	Avg. states	Avg. time (s)	Property	Points-to	Enum/Bool	Constant	Other
Overview	-	-	-	4	92.00	211.75	12	8	9	2	7
Alchemist	2	19.00	29.00	1	38.00	154.00	4	2	-	-	-
Brickblock	3	65.00	663.67	3	185.67	191.67	9	-	3	-	1
Crowdsale	-	-	-	9	73.67	261.33	23	27	-	18	-
ERC20	-	-	-	9	48.00	158.33	24	-	-	-	-
ICO	-	-	-	8	283.63	6817.00	31	72	-	48	-
Mana	-	-	-	4	441.75	41409.25	10	16	1	2	2
Melon	-	-	-	16	159.38	408.38	50	-	-	48	-
MRV	4	109.00	1075.75	1	128.00	887.00	3	-	-	1	-
PolicyPal	-	-	-	4	667.50	20773.50	9	8	-	12	-
VUToken	-	-	-	5	123.00	715.40	11	5	-	33	-
Zebi	3	41.33	77.67	2	116.50	13604.00	3	3	-	2	-
Zilliqa	5	71.80	94.40	-	-	-	-	-	-	-	-
Average		18.01	114.15		35.00	1296.84	2.86	2.14	0.20	2.52	0.15
Sum	17			66							

TABLE III: Experimental results of VERX's property verification using inductive reasoning and delayed abstraction.

		VERX					Oyente					Mythril					Manticore				
Benchmark	Tests	TP	TN	FN	FP	T/O	TP	TN	FN	FP	T/O	TP	TN	FN	FP	T/O	TP	TN	FN	FP	T/O
Assertions	19	11	7	0	1	0	0	3	10	6	0	7	4	4	3	1	6	7	5	0	1
Arithmetic	22	6	16	0	0	0	5	7	1	9	0	6	13	0	3	0	6	16	0	0	0

TABLE IV: Comparison of VERX's symbolic execution engine with state-of-the-art symbolic tools.

Finally, 10 of the projects do not require any custom predicates to verify all their properties. This indicates that most properties can be verified with few extra predicates which are extracted automatically. The remaining 3 projects do require few predicates classified as *Other*. Example predicates are `Crowdsale.raised < Crowdsale.goal` and `sum(Escrow.deposits)== Crowdsale.raised`. We note that such predicates often appear in function preconditions (defined in `require` statements), indicating that they can be easily extracted automatically.

C. Importance of Derived Predicates

Most of the properties in our benchmarks are non-inductive and therefore require additional predicates to verify. We inspected the relevance of the predicates derived automatically by VERX and report our findings below.

Points-to predicates. Points-to predicates are essential to prove that the bundle contracts are effectively external callback free (EECF). Calls to bundle contracts are often followed by changes to storage and thus do not satisfy the first condition imposed by VERX on external calls (see Section VIII). For example, consider function `invest()` of the crowdsale (Fig. 2)

which calls the escrow contract. Without additional predicates, the variable `escrow` is abstracted away, meaning that it can store the address of any (including an external) contract. Under this abstraction, VERX fails to verify the specification because the function writes to variable `raised` after the call. To avoid this false alarm, VERX automatically adds the points-to predicate: `Crowdsale.escrow == Escrow` and proves that variable `escrow` always points to the escrow (a bundle contract), and by doing so avoids aborting the verification process.

Boolean/enumerated predicates. For our overview example, VERX also adds the three predicates

$$\{\texttt{Escrow.state} == X \mid X \in \{\texttt{OPEN}, \texttt{SUCCESS}, \texttt{REFUND}\}\}.$$

These predicates are needed to capture the possible state transitions of the escrow. Without them, VERX analyzes spurious traces which violate the properties. Concretely, VERX fails to prove that the escrow cannot transition from a state where `Escrow.state == REFUND` holds to a state where `Escrow.state == SUCCESS` holds (and vice versa) and, in turn, fails to verify property $\varphi_{\mathbf{R2}}$.

D. VERX's Symbolic Execution Engine

Finally, we compare VERX's symbolic execution engine to three other state-of-the-art engines: Oyente [47], Mythril v0.19.3 [50], and Manticore v0.1.9 [2]. We collected 41 tests from the smart contract security project [5], where security experts maintain a collection of vulnerable smart contracts. Our benchmark consists of two kinds of tests: *assertions* and *arithmetic*, where the tests check for failed assertions and, respectively, over- and under-flows.

Table IV shows the results. For each of the four systems, we report: detected (TP), missed (FN), falsely reported (FP), and correctly unreported (TN) violations. In our tests, we used a timeout (T/O) of one hour.

Results indicate that VERX outperforms other symbolic execution tools in terms of precision. VERX has no false negatives and only 1 false positive. The test for which VERX reports a FP is a synthetic example that checks if the remaining gas decreases during execution, and VERX over-approximates gas-mechanics, which is sound for verification.

X. RELATED WORK

We now describe the works that are closely related to ours.

Predicate abstraction. Predicate abstraction [35] has been shown successful to verify safety properties. SLAM [14], [15] automatically constructs a boolean program from a C program to verify properties. BLAST [39], [40] increases precision by adding predicates derived from the proofs of infeasible traces. SatAbs [17], MAGIC [23], Murphy [30], and Java PathFinder [29] also build on predicate abstraction for verification and differ mostly in the properties and programs analyzed. Other works show how to apply refinement during abstraction, e.g., the authors of [49] describe refinement by partitioning traces. Our work is inspired by the above approaches. In contrast, it is tailored to contracts and applies abstraction only at the end of transactions.

Symbolic execution. Symbolic execution has been shown successful in finding property violations of programs [12], [22], [45], [53], [63], [67]. To mitigate path explosion due to loops and recursive calls, different heuristics have been suggested: favoring paths that may lead to uncovered code [22], constraining search depth [53], using fitness functions [67], chopping functions [63], function and loop summaries to reuse repetitive computations [10], [33], [34], and using concrete inputs to select symbolic paths [11], [56], [57], [60], [70]. The authors of [69] use symbolic execution to prune paths irrelevant when verifying regular properties, which is sound under input independency assumptions. To reduce paths via subsumption, the authors of [13] combine symbolic execution with abstraction to analyze programs that manipulate arrays and lists. In the context of contracts, there are different symbolic engines for Ethereum [2], [24], [46], [47], [50], [51]. Some of these support multiple transactions and employ heuristics to guide the path exploration. In contrast, our symbolic engine is more precise, as shown in our evaluation, and handles hash collisions more efficiently.

Analysis of temporal properties. Many works have studied analysis of temporal properties. In [55], the authors show an approach to test for violations to past LTL formulas. In [16], the authors show a verification of LTL formulas for UML models. T2 [21] is a system to verify temporal properties over LLVM which supports linear integer arithmetic programs. E-HSF [18] verifies existential CTL formulas by representing them as existentially quantified Horn-like clauses, and using a counterexample-guided approach to solve the clauses. Typestates [61] enable one to express correct usage rules of class operations or protocols. Some of our evaluated properties can be expressed as typestates.

Static analysis and verification of smart contracts. Declarative static analysis tools [20], [36], [64], [65] use Datalog to identify generic security vulnerabilities in Ethereum contracts. These tools focus on data-flow and gas-related vulnerabilities. Slither [4] and SmartCheck [62] identify vulnerabilities on the abstract-syntax tree of the contract's source code. The analysis of Grossman et al. [38] targets vulnerabilities due to callbacks in contracts. In contrast to our work, these approaches do not support the verification of temporal properties. In [43], Hirai defines a formal model for EVM in the Lem language. Formal EVM semantics have been defined also defined for F* [37], the K framework [42], and Isabelle/HOL [9]. These approaches provide formal guarantees while being precise (no false positives). Some of these frameworks are non-trivial to automate. In contrast, we target automated verification of temporal properties. Analysis of temporal properties of smart contracts has also been considered in [59], where authors show how to manually encode such properties and verify them with Coq. In contrast, we show an automatic approach and provide an end-to-end tool to analyze temporal safety properties.

XI. Conclusion

We presented VERX, the first verifier that can automatically prove temporal safety properties of Ethereum smart contracts. The verifier is based on a careful combination of three ideas: reduction of temporal safety verification to reachability checking, an efficient symbolic execution engine used to compute precise symbolic states within a transaction, and delayed abstraction which approximates symbolic states at the end of transactions into abstract states. Delayed abstraction allows the verifier to reduce the potentially unbounded concrete state space of the contact into a finite, bounded representation, while still maintaining precision (thanks to employing precise reasoning during transactions). We demonstrated that VERX is practical and automatically proves 83 properties over 12 real-world Ethereum projects. Based on these experiments, we believe VERX is an effective system for verifying custom functional properties of smart contracts.

Acknowledgments

The authors would like to thank Hubert Ritzdorf from ChainSecurity AG for the valuable feedback on the specifics of the Ethereum Virtual Machine. We also thank the anonymous reviewers for their constructive comments.

References

[1] The dao hack explained: Unfortunate take-off of smart contracts, 2018. Available from: https://medium.com/@ogucluturk/.

[2] Manticore, 2018. Available from: https://github.com/trailofbits/manticore.

[3] Openzeppelin library, 2018. Available from: https://github.com/OpenZeppelin/openzeppelin-solidity.

[4] Slither, 2018. Available from: https://github.com/trailofbits/slither.

[5] Smart contract security project, 2018. Available from: https://github.com/SmartContractSecurity/SWC-registry/.

[6] Solidity language documentation, 2018. Available from: https://solidity.readthedocs.io.

[7] Formal verification of multicollateral dai, 2019. Available from: https://github.com/dapphub/k-dss/.

[8] Solidity docs: Access to external variables, functions and libraries, 2019. Available from: https://solidity.readthedocs.io/en/v0.5.9/assembly.html#access-to-external-variables-functions-and-libraries.

[9] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *International Conference on Certified Programs and Proofs, CPP*. ACM, 2018.

[10] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 367–381, 2008.

[11] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Symposium on the Foundations of Software Engineering, FSE*, 2012.

[12] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 134–138, 2007.

[13] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer, STTT*, pages 53–67, 2009.

[14] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213. ACM, 2001.

[15] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Symposium on Principles of Programming Languages, POPL*, pages 1–3. ACM, 2002.

[16] Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi. Efficient scalable verification of LTL specifications. In *International Conference on Software Engineering, ICSE*, pages 711–721. ACM/IEEE, 2015.

[17] Gérard Basler, Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. satabs: A bit-precise verifier for C programs - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 552–555, 2012.

[18] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified horn clauses. In *Computer Aided Verification, CAV*, pages 869–882, 2013.

[19] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Selecta formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, pages 234–245, 1975.

[20] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, 2018.

[21] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: temporal property verification. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 387–393, 2016.

[22] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 209–224, 2008.

[23] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering, ICSE*, pages 385–395. ACM/IEEE, 2003.

[24] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. scompile: Critical path identification and analysis for smart contracts. *CoRR*, 2018.

[25] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference Computer Aided Verification, CAV*, pages 154–169, 2000.

[26] Byron Cook, Eric Koskinen, and Moshe Vardi. Temporal property verification as a program analysis task. In *International Conference Computer Aided Verification, CAV*. Springer Berlin Heidelberg, 2011.

[27] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Symposium on Principles of programming languages, POPL*, pages 238–252. ACM, 1977.

[28] Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 84–104, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[29] Jakub Daniel, Pavel Parizek, and Corina S. Pasareanu. Predicate abstraction in java pathfinder. *Software Engineering Notes*, pages 1–5, 2014.

[30] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification, CAV*, pages 160–171, 1999.

[31] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 337–340, 2008.

[32] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *SIGPLAN Notices*, pages 191–202. ACM, 2002.

[33] Patrice Godefroid. Compositional dynamic test generation. In *Symposium on Principles of Programming Languages, POPL*, pages 47–54. ACM, 2007.

[34] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 23–33, 2011.

[35] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, CAV*, pages 72–83, 1997.

[36] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. ACM, 2018.

[37] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust, POST*, pages 243–269, 2018.

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

[39] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Symposium on Principles of Programming Languages, POPL*, pages 232–244. ACM, 2004.

[40] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages, POPL*, pages 58–70. ACM, 2002.

[41] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 463–492, 1990.

[42] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *Computer Security Foundations Symposium (CSF)*, 2018.

[43] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Security*. Springer International Publishing, 2017.

[44] Theodoros Kasampalis, Dwight Guth, Brandon Moore, Traian Serbanuta, Virgil Serbanuta, Daniele Filaretti, Grigore Rosu, and Ralph Johnson. Iele: An intermediate-level blockchain language designed and implemented using formal semantics. Technical report, July 2018.

[45] James C King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

[46] Johannes Krupp and Christian Rossow. Teether: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security*, 2018.

[47] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Conference on Computer and Communications Security, CCS*, 2016.

[48] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer, 1995.

[49] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming, ESOP*, pages 5–20, 2005.

[50] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 2018.

[51] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Annual Computer Security Applications Conference, ACSAC*, pages 653–663. ACM, 2018.

[52] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, pages 631–653, 1979.

[53] Corina S. Pasareanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *International Conference on Automated Software Engineering, ASE*, pages 179–180. ACM/IEEE, 2010.

[54] Grigore Roşu. On safety properties and their monitoring. *Scientific Annals of Computer Science*, pages 327–365, 2012.

[55] Grigore Rosu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties: This time with calls and returns. In *International Workshop on Runtime Verification, RV*, pages 51–68, 2008.

[56] Koushik Sen. Concolic testing. In *International Conference on Automated Software Engineering ASE*, pages 571–572. IEEE/ACM, 2007.

[57] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *International Symposium on Foundations of Software Engineering, FSE*, pages 263–272, 2005.

[58] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 478–493. Springer, 2017.

[59] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 323–338, 2018.

[60] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium, NDSS*, 2016.

[61] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, pages 157–171, 1986.

[62] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB*, 2018.

[63] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering, ICSE*, pages 350–360. ACM/IEEE, 2018.

[64] Petar Tsankov. Security analysis of smart contracts in datalog. In *Leveraging Applications of Formal Methods, Verification and Validation*. Springer International Publishing, 2018.

[65] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Conference on Computer and Communications Security, CCS*, 2018.

[66] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[67] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *International Conference on Dependable Systems and Networks, DSN*, pages 359–368, 2009.

[68] Zheng Yang and Hang Lei. Formal process virtual machine for smart contracts verification. *CoRR*, abs/1805.00808, 2018.

[69] Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. Symbolic verification of regular properties. In *International Conference on Software Engineering, ICSE*, pages 871–881. IEEE/ACM, 2018.

[70] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. Regular property guided dynamic symbolic execution. In *International Conference on Software Engineering, ICSE*, pages 643–653. IEEE/ACM, 2015.