Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid

Keerthi Nelaturu*, Anastasia Mavridou†, Andreas Veneris*, Aron Laszka‡
* University of Toronto † SGT Inc. / NASA Ames Research Center ‡ University of Houston

***Abstract*—Smart contracts enable the creation of decentralized applications which often handle assets of large value. These decentralized applications are frequently built on multiple interacting contracts. While the underlying platform ensures the correctness of smart contract execution, today developers continue struggling to create functionally correct contracts, as evidenced by a number of security incidents in the recent past. Even though these incidents often exploit contract interaction, prior work on smart contract verification, vulnerability discovery, and secure development typically considers only individual contracts. This paper proposes an approach for the *correct-by-design development* and *deployment* of multiple interacting smart contracts by introducing a graphical notation (called deployment diagrams) for specifying possible interactions between contract types. Based on this notation, it later presents a framework for the automated verification, generation, and deployment of interacting contracts that conform to a deployment diagram. As an added benefit, the proposed framework provides a clear separation of concerns between the internal contract behavior and contract interaction, which allows one to compositionally model and analyze systems of interacting smart contracts efficiently.**

***Index Terms*—Smart Contract, Verification, CAD, Solidity, Ethereum.**

I. Introduction

Blockchain technology has received significant attention in recent years from academia and industry due to its ability to provide open, decentralized, and trustworthy platforms of computation. While undoubtedly the most widely used application of blockchains today is remittance of cryptocurrencies (e.g., BTC on Bitcoin network), blockchains can also be used as trustworthy decentralized mediums for general-purpose trustless computation in the form of smart contracts.

As with any software implementation, smart contracts may suffer from subtle and surprising bugs made by developers. These bugs present potential threats to the security of smart contracts by allowing attackers to maliciously extract currency from contracts or to even destroy the contracts in certain cases. Recent notable security incidents are the "DAO attack" [1], in which around 3.6 million Ether was transferred to the perpetrator (valued today at around $747 million), and the "Parity Wallet hack" [2], in which around 514 thousand Ether was permanently frozen (valued today at around $107 million).

Motivated by such security issues and incidents, a number of tools have been proposed for vulnerability discovery (e.g., [3], [4], [5]), formal verification (e.g., [6], [7], [8]), correct-by-design construction (e.g., [9], [10]), etc. to aid developers in creating secure contracts. A common limitation of existing tools is that they typically focus on the analysis or development of a single contract in isolation from other contracts with which it may have to interact once it is deployed. However, in practice, most decentralized applications are built on multiple interacting smart contracts, and exploits often involve more than one of them. For example, in the so-called "DAO attack," the perpetrator exploited a re-entrancy vulnerability in the `DAO` contract that involved function calls to other contracts.

In this paper, we propose an end-to-end framework for the *correct-by-design development* and *deployment* of *multiple interacting smart contracts* for Ethereum's Solidity. Our work builds upon the VeriSolid open-source framework [10], which supports the correct-by-design development of standalone contracts. Contrary to other tools that support vulnerability discovery and analysis of existing smart contracts, the idea behind VeriSolid is to help users develop correct-by-design smart contracts through iterative analysis of high-level models and code generation. In particular, VeriSolid allows developers to graphically design a smart contract as a transition system, perform model checking, and generate functionally equivalent Solidity code based on formally defined operational semantics.

In detail, the contributions of this paper are as follows:

- We extend the VeriSolid operational semantics to formally capture interaction of transition systems.
- We propose a graphical notation, called Solidity Deployment Diagrams, for specifying permitted interactions between smart contracts.
- We introduce an approach for the formal verification of a system of interacting smart contracts.
- We extend VeriSolid Solidity code generation to support the generation of multiple interacting contracts.
- We provide a deployment framework of the generated smart contracts on the Ethereum blockchain.

The remainder of this paper is organized as follows. Section II provides background information on VeriSolid. Section III introduces our novel approach for multiple interacting contracts. Section IV describes the transformation of smart contract Solidity code into verifiable models. Section V introduces our graphical notation for modeling contract interaction. Section VI presents our verification methodology. Section VII discusses examples of typical vulnerabilities that can be prevented or detected using our framework. Section VIII discusses related work and Section IX concludes the paper.

978-1-7281-6680-3/20/$31.00@2020 IEEE

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

II. Background: VeriSolid Framework

VeriSolid enables developers to *(i)* specify smart contracts as transition systems, *(ii)* verify these systems individually, and *(iii)* generate functionally equivalent Solidity code from them [10]. The states of a transition system model the various states of a contract (e.g., different stages of a secret ballot vote or of a blind auction), while the transitions between these states model functions that may be externally called to change the state of the contract. For the sake of completeness, we provide a brief overview of the formal syntax and verification approach of VeriSolid [10].

VeriSolid allows developers to specify the actions that are performed by transitions (i.e., function bodies) using a Turing-complete subset of Solidity statements denoted by $\mathbb{S}$. Further, let $\mathbb{T}$ denote the set of Solidity data types, $\mathbb{I}$ denote the set of valid Solidity identifiers, $\mathbb{D}$ denote the set of Solidity events and custom data type definitions, $\mathbb{E}$ denote the set of Solidity expressions, and $\mathbb{C}$ ($\subseteq \mathbb{E}$) denote the set of Solidity expressions that do not have any side effects (apart from consuming gas or raising an exception). Then, a transition system for modeling a smart contract is a tuple $(D, S, S_F, s_0, a_0, a_F, V, T)$:

- $D \subset \mathbb{D}$ is a set of custom event and type definitions;
- $S \subset \mathbb{I}$ is a finite set of states;
- $S_F \subset S$ is a set of final states;
- $s_0 \in S$, $a_0 \in \mathbb{S}$ are the initial state and action;
- $a_F \in \mathbb{S}$ is the fallback action;
- $V \subset \mathbb{I} \times \mathbb{T}$ contract variables (i.e., variable names and types);
- $T \subset \mathbb{I} \times S \times 2^{\mathbb{I} \times \mathbb{T}} \times \mathbb{C} \times (\mathbb{T} \cup \emptyset) \times \mathbb{S} \times S$ is a transition relation, where each transition $t \in T$ includes: transition name $t^{name} \in \mathbb{I}$; source state $t^{from} \in S$; parameter variables (i.e., arguments) $t^{input} \subseteq \mathbb{I} \times \mathbb{T}$; transition guard $g_t \in \mathbb{C}$; return type $t^{output} \in (\mathbb{T} \cup \emptyset)$; action $a_t \in \mathbb{S}$; destination state $t^{to} \in S$.

A contract can have at most one constructor represented by the initial action a_0. After the constructor returns, the contract is in initial state s_0. A contract can also have at most one unnamed function, which is called the fallback function and represented by the fallback action a_F. Other functions are represented by transitions T. A transition $t \in T$ expects a set of function arguments t^{input}, executes its action a_t if the contract is in the source state t^{from} and the guard condition g_t is met, and moves the contract into destination t^{to} upon successful execution (i.e., no exceptions raised). For the interested reader, [10] provides formal operation semantics that allow the mechanics and formalism presented here.

In VeriSolid, formal verification is essential to check the behavioral correctness of the system under design. While alternative approaches (such as simulation or testing) rely on the selection of appropriate test input stimulus for a predetermined coverage of the program's control flow, formal verification (e.g., by model checking) guarantees full coverage of execution paths for all possible inputs. Thus, it provides a *rigorous* way to assert (or disprove) that a system model meets a set of predefined properties. VeriSolid uses `nuXmv`, a symbolic model checker for transition systems. To enable `nuXmv` to consider transition actions (i.e., Solidity statements in function bodies), VeriSolid transforms each transition action into a series of internal states and internal transitions that represent the control flow of the action. Specifically, it applies an *augmentation algorithm* that recursively replaces compound statements with series of inner transitions and states, and recursively replaces control-flow statements (e.g., selection and loop statements) with inner transitions and states modeling the possible execution traces. The output of this augmentation algorithm is a transition system with only simple statements (i.e., expression statements, variable declarations, and event emissions) as actions. The augmentation also considers the possibility of a function encountering an exception and reverting its execution.

III. Proposed Workflow

We extend VeriSolid with a novel approach for the correct-by-design development and deployment of multiple interacting smart contracts. We provide an open-source, web-based implementation [11] which allows the collaborative development of Ethereum contracts with built-in version control, which enables branching, merging, and history viewing. We also extend VeriSolid to support Solidity v0.5.

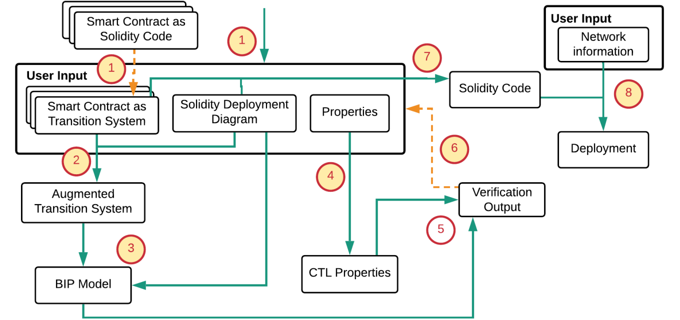

Fig. 1. Design, verification, and deployment workflow

Figure 1 shows the steps (numbered arrows) of the design, verification, and deployment workflow of our approach that extends the original VeriSolid framework. Circled numbers with shaded background represent steps that do not exist or are significantly different than in the previous version of VeriSolid. Solid arrows represent steps that are mandatory, while dashed arrows represent optional steps. In Step 1, the developer provides input, which consists of:

- Contract specifications containing: 1) a graphically specified transition system and 2) variable declarations, actions, and guards specified in Solidity. Alternatively, a developer may import the code of an existing Solidity contract and the system automatically creates the corresponding transition system, as detailed in Section IV.
- A list of properties to be verified. These properties can be expressed in predefined natural-language like templates. Properties are extended to enable specifying requirements on multiple interacting smart contracts (Section VI).
- A graphically specified Solidity Deployment Diagram that contains contract types and their associations. The associations specify which contracts have references to which other contracts (Section V).

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Transition System Generation

Input: Solidity source code
Output: model $(D, S, S_F, s_0, a_0, a_F, V, T)$
1: model ← **call** AddStatesTransitions(source) [Algorithm 2]
2: model ← **call** AddModifiers(source, model)
3: **return** model [Algorithm 3]

The verification loop (Steps 2 to 6) starts at the next step. Step 2 is automatically executed to generate the augmented contract models based on the transition systems and the deployment diagram information. Next, in Step 3, the Behavior-Interaction-Priority (BIP) model of the interacting smart contracts is automatically generated. Similarly, in Step 4, the specified properties that may involve multiple interacting contracts are automatically translated to Computational Tree Logic (CTL). The model can then be verified for deadlock freedom or other properties using tools from the BIP tool-chain [12] or `nuXmv` [13] (Step 5). If the required properties are not satisfied by the model (depending on the output of the verification tools) the input can be refined by the developer (Step 6) and analyzed anew. Finally, when the developer is satisfied with the design, i.e., all specified properties are satisfied, the equivalent Solidity code of the interacting contracts is automatically generated in Step 7. At this point (Step 8), the user may use our deployment plugin to correctly deploy the verified contracts onto a blockchain network.

Scope of Supported Operations: To keep verification computationally tractable for multiple contracts, we impose certain restrictions on the interactions between the contracts. We believe that these restrictions are plausible and realistic, as they enable the correct-by-design implementation of a wide range of decentralized applications. In particular, we make the following assumptions on contract interactions:

- We allow only the following interactions between generated Solidity contracts: high-level functions calls (i.e., *contract*`.`*function*`(`*arg1*, *arg2*, ...`)`), transfers (i.e., *address*`.transfer(`*amount*`)`), and a custom high-level delegation mechanisms, which is implemented as a wrapper around the low-level `delegatecall` and raises an exception upon failure;
- We do not allow re-entrancy for external function calls, that is, if contract x calls another contract, then before this call finishes, no other function may be invoked in contract x; and,
- We allow only non-recursive internal function calls, that is, if function f is invoked, then it may not be invoked again before the current invocation finishes.

IV. Transition System Generation

In Step 1, a developer may import the Solidity code of an existing contract and use our automatic mechanism, described in Algorithm 1, to generate a corresponding transition system. Algorithm 1 uses the following two sub-algorithms: `AddStatesTransitions` (Algorithm 2) and `AddModifiers` (Algorithm 3).

Algorithm 2: Add States and Transitions

Input: Solidity source code
Output: model $(D, S, s_0, a_0, a_F, V, T)$
1: $D \leftarrow \emptyset$, $S \leftarrow \{s\}$, $S_F \leftarrow \emptyset$, $a_0 \leftarrow \{\}$, $a_F \leftarrow \{\}$, $V \leftarrow \emptyset$, $T \leftarrow \emptyset$
2: **for** *event* **in** source:
3: $D \leftarrow D$ + *event*
3: **end for**
4: **for** contract variable @*type* @*identifier* **in** source:
5: $V \leftarrow V$ + (@*identifier*, @*type*)
6: **end for**
7: **if** `constructor` **in** source:
8: $a_0 \leftarrow$ constructor body
9: **end if**
10: **if** fallback `function ()` **in** source:
11: $a_F \leftarrow$ fallback body
12: **end if**
13: **for** function @*type* `function` @*name*(@*input*) **in** source:
14: $T \leftarrow T+$ transition t, where $t^{name} = @name$, $^{from} = s$, $t^{to} = s$, $t^{input} = @input$, $g_t \leftarrow$ `true`, $t^{output} = @type$, and $a_t =$ function body
15: **end for**

Algorithm 3: Add Modifiers

Input: Solidity source code, model $(D, S, s_0, a_0, a_F, V, T)$
Output: model $(D, S, s_0, a_0, a_F, V, T)$
1: *modifiers* ← $\emptyset$
2: **for** `modifier` @*mname* @*mbody* **in** source:
3: *modifiers* ← *modifiers* + (@*mname*, @*mbody*)
4: **end for**
5: **for** $t \in T$:
6: **for** @*mname* **in** modifiers of `function` t^{name}:
7: @*mbody* ← **find** @*mname* **in** *modifiers*
8: **if** @*mbody* specifies side-effectless condition c:
9: $g_t \leftarrow g_t \wedge c$
10: **else**
11: $a_t \leftarrow a_t$ extended with @*mbody*
12: **end if**
13: **end for**
14: **end for**

`AddStatesTransitions` creates the states, transitions, initial action, and fallback actions. `AddModifiers` extends the transitions based on the modifiers of the corresponding functions. After the extension, the generated transitions may contain guard conditions depending on whether the corresponding functions use Solidity modifiers. Note that not all modifiers can be converted into guards. The modifiers that our framework converts into guards must follow the syntax below, i.e., they must include `require` and/or `if` statements and their execution must come before the body of the corresponding function:

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

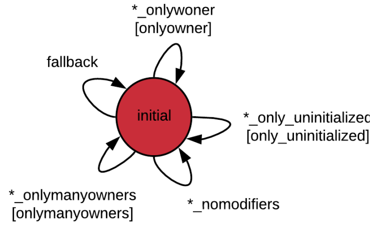

Fig. 2. Transition system of WalletLibrary

$$\langle guard_modifier \rangle ::=$$

```
modifier @identifier ( (@type @identifier
                        (, @type @identifier) * ) * )
                        { (if (@expression)) * |
                         (require (@expression) ; )*
                        | _; }
```

If there are multiple `if`, `require` statements in the modifier, we append all the expressions with a logical `&&` operator forming a conjunction. Once a transition systems is generated, the developer may update the transition systems by adding, removing, or modifying states, transitions, etc.

As an example, consider an earlier version of the Parity multisignature wallet, which became famous as the victim of one of the largest Ethereum security incidents to date [2]. A single instance of the Parity `WalletLibrary` contract was deployed and used as a library by a number of Parity `Wallet` contracts, which heavily relied on the code of the library since they simply delegated most function calls to the library.

We automatically generated the transition systems of `Wallet` and `WalletLibrary` by importing their source code, which is available on Etherscan [14] (350 lines of code for `WalletLibrary` and 60 lines of code for `Wallet`).

From the source code of `WalletLibrary`, our framework generated the model in Figure 2. To increase readability of the model, our framework groups functions that use the same modifiers—the guards of the transition system—as follows:

- `*_onlyowner` contains the `execute` and `underLimit` functions;
- `*_onlyinitialized` contains the `initmultiowned`, `initDayLimit`, `initWallet` functions;
- `*_nomodifier` contains the `getOwner`, `isOwner`, `hasConfirmed`, `revoke`, `create`, `confirm`, `confirmCheck`, `reorganizeOwners`, `clearPending`, `today` functions;
- `*_onlymanyowners` contains the `changeOwner`, `addOwner`, `removeOwner`, `changeRequirement`, `setDayLimit`, `resetSpentToday`, `kill` functions.

Similarly, from the source code of `Wallet`, our framework generated the model shown in Figure 3. The initial transition `Wallet` corresponds to the constructor of the contract.

Each generated transition system contains a single state. The `Initial` state of `WalletLibrary` represents initial state of the contract after creation. Note that the contract

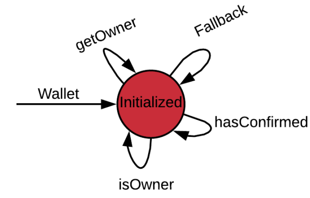

Fig. 3. Transition system of Wallet

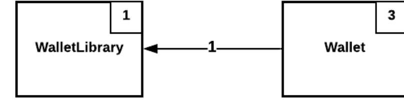

Fig. 4. SDD of Parity Wallet

does not have a constructor, so all variables are zero. The `Initialized` state of `Wallet` represents the main state of the contract, which is entered once the owners are set up.

One of the functions of `WalletLibrary`, named `initMultiOwned` (generated as a transition), played a crucial role: it was invoked from the constructor of a wallet using delegation to set up the owners of the wallet. However, due to a design mistake, anyone could call this function directly in an instance of `WalletLibrary`, thereby taking ownership of the library itself. Once someone has taken ownership of the `WalletLibrary` instance, they could easily destroy it by calling its `kill` function. The destruction of the library meant that all wallets relying on it were deadlocked, i.e., lost almost all of their functionality, including the ability to send or withdraw funds. This deadlock resulted from the interaction between the `Wallet` and the `WalletLibrary`. At the time of this writing, the amount of Ether "deadlocked" in these wallets is approximately $107 million.

V. Modeling Smart Contract Interactions

To specify contract interaction rules for verification, the developer must provide a *Solidity Deployment Diagram* (SDD) in Step 1. In this section, we focus on the specification of deployment information between contract types based on the concept of *association*.

Figure 4 shows the SDD of Parity Wallet example, which contains two contract types: `WalletLibrary` and `Wallet`. Each of the contract types has an associated natural number, namely cardinality, that defines the number of instances that must be deployed for each contract type, e.g., 3 for `Wallet` and 1 for `WalletLibrary`. Additionally, the SDD contains an arrow associating each instance of `Wallet` with the single instance of `WalletLibrary`. This means that each `Wallet` instance must have a reference to the `WalletLibrary` instance, which can be used for delegating or calling functions.

We now define formally the aforementioned concepts.

Definition 1. *An* SDD $\langle \mathcal{T}, n, \mathcal{A} \rangle$ *consists of a set of contract types* $\mathcal{T} = \{T_1, \ldots, T_K\}$*; an associated cardinality function* $n : \mathcal{T} \rightarrow \mathbb{N}$*, where* $\mathbb{N}$ *is the set of natural numbers (we will abbreviate* $n(T_i)$ *to* n_i *to simplify the notation); and a set of deployment associations* $\mathcal{A} = \{A_1, \ldots, A_l\}$ *of the form* $A =$

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

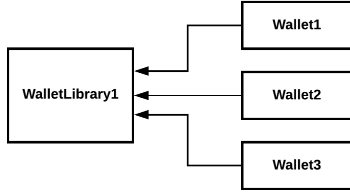

Fig. 5. Example deployment of Parity Wallet

$(\langle a_s, a_t \rangle, d)$, where $a_s, a_t \in \mathcal{T}$ are respectively the source and target of the deployment association, and $d \in \mathbb{N}_{>0}$ is the degree of the association.

The degree of an association constrains the number of associations attached to each instance of the source contract type (a_s). The number of associations attached to each instance of the target contract type (a_t) is equal to the cardinality n_{a_s} of the source contract type multiplied by the degree d. Notice that deployment associations are binary, i.e., they involve exactly two contract types.

The degree of an association must be equal to or less than the cardinality of a_t, otherwise the SDD is invalid. Our framework checks and notifies the developer of invalid SDDs. In the Parity Wallet, we have only two contract types with a single association between them but in other cases a single contract may be referencing several other contracts. Manually adding this information may be an error prone task. Our framework provides a high-level, diagrammatic view of the architecture of the system, which gives the developer a clear idea of the involved contracts and how they interact. Based on this information, our framework automatically generates in Solidity the references between smart contracts during the Solidity code generation (Step 7 in Figure 1).

Next, we formally define a deployment instance (Definition 2) and the Deployment Semantics, which describe conditions that a deployment instance must satisfy in order to conform to a given SDD. For instance, the deployment shown in Figure 5 conforms to the SDD of Figure 4.

Definition 2. *A* deployment *is a pair $\langle \mathcal{C}, \gamma \rangle$, where $\mathcal{C}$ is a set of contract instances and γ is a deployment configuration, i.e., a set of binary associations among the contract instances in $\mathcal{C}$. Each contract instance $C \in \mathcal{C}$ is specified as a pair $\langle T, \mathbf{v} \rangle$ of contract type $T \in \mathcal{T}$ and constructor parameter values $\mathbf{v}$, i.e., C is instantiated from type T with parameters $\mathbf{v}$.*

Definition 3. *[Deployment Semantics] A deployment $\langle \mathcal{C}, \gamma \rangle$* conforms *to an SDD $\langle \mathcal{T}, n, \mathcal{A} \rangle$ if 1) for each $i \in [1, k]$, the number of contracts of type C_i in $\mathcal{C}$ is equal to n_i and 2) for each association $A \in \mathcal{A}$ and instance $C_i \in \mathcal{C}$ such that C_i is of type a_s, there exist exactly d instances $C_j \in \mathcal{C}$ of type a_t such that $(C_i, C_j) \in \gamma$.*

The second condition can be written formally as follows:

$$\forall (\langle a_s, a_t \rangle, d) \in \mathcal{A}, C_i \in \mathcal{C} : \; C_i \text{ is of type } a_s \Rightarrow$$
$$d = |\{C_j | C_j \in \mathcal{C} \wedge C_j \text{ is of type } a_t \wedge (C_i, C_j) \in \gamma\}|$$

VI. Verifying Interacting Smart Contracts

Once a developer has provided the required input (Step 1 in Figure 1), the verification loop begins. Steps 2 and 3 include the automatic generation of the augmented transition systems and BIP model. To support the augmentation process, we have extended the set of supported statements $\mathbb{S}$ by including `selfdestruct(` $@expression$ `);` and our custom, high-level delegation invocation $@expression$`.delegate.`$@identifier$`(` $\big(@expression($`,` $@expression)*\big)?$`)`. Next, we present the necessary modeling concepts of the BIP component framework [12].

a) Modeling with BIP: Systems are modeled in BIP [15], [16] by superposing the Behavior, Interaction, and Priority layers. The *Behavior* layer consists of a set of components represented by transition systems. Each component transition is labeled by a *port*. Ports form the interface of a component are used for interaction with other components. Additionally, each transition may be associated with a set of *guards* and a set of *actions*. A guard is a predicate on variables that must be true to allow the execution of the associated transition. An action is a computation triggered by the execution of the associated transition.

Component interaction is described in the *Interaction* layer. A BIP interaction is a non-empty set of ports that synchronize (i.e., their corresponding transitions are jointly executed). We represent component interaction with connectors between component ports. In the context of smart contracts, we use BIP interactions to model: 1) function calls between different contracts and 2) call delegations. We omit the explanation of the *Priority* layer since we do not use it in our contract models.

b) Operational Semantics of Interacting Smart Contracts: To apply formal verification, we define the operational semantics of smart contract interaction in the form of Structural Operational Semantics (SOS) rules [17]. Next, we present rules of the normal execution of a transaction and a function. Due to space limitations, we have included all additional rules that capture both normal execution and exceptions for call delegation and nested function calls in [18]. We let Ψ denote the state of the ledger, which includes account balances, values of state variables in all contracts, number and timestamp of the last block, etc. We let $\mathbf{s}$ the current states of contracts of the system. During the execution of a function, the execution state $\sigma = (\Psi, M, \mathbf{s}, \kappa)$ also includes the memory and stack state M, and the set of destroyed contracts κ. To handle return statements and exceptions, we also introduce an execution status, which is E when an exception has been raised, $R[v]$ when a return statement has been executed with value v (i.e., `return` v), and N otherwise. Finally, we let $\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle \hat{\sigma}, R[v] \rangle$ signify that the evaluation of a Solidity expression Exp in execution state σ yields value v. On the other hand, $\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle \hat{\sigma}, E \rangle$ signifies that the evaluation has resulted in an exception.

In our model, an externally owned account initiates a transaction by providing a contract instance $i \in \mathcal{C}$ with a function name *name* $\in \mathbb{I}$ and a list of parameter values $v_1, v_2, \ldots$. The transaction invokes the function in the current ledger and contract states Ψ and $\mathbf{s}$, which results in changed ledger and contract states Ψ' and $\mathbf{s}'$ as well as a set of contracts κ that have selfdestructed during execution (see rule FUNC). The transaction changes the states of these contracts $j \in \kappa$ to a special state $s''_j =$ *destroyed*, and then makes the

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

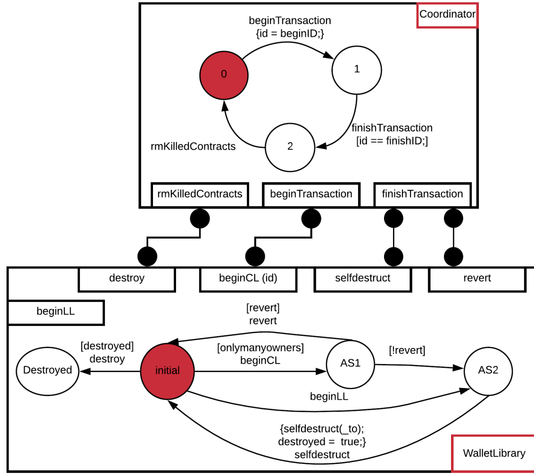

Fig. 6. `WalletLibrary`: Global Coordinator & augmented component part

new ledger and contract states Ψ' and $\mathbf{s}''$ permanent. This normal execution is captured by the TRANS rule:

$$\text{TRANS}\ \frac{\begin{array}{c}\langle(\Psi,\mathbf{s},\emptyset), i.\mathit{name}\,(v_1,v_2,\ldots)\rangle \to \\ \langle(\Psi',\mathbf{s}',\kappa),x\rangle,\quad x\in\{N,R[v]\} \\ \forall j\in\kappa: s''_j = \mathit{destroyed},\quad \forall j\notin\kappa: s''_j = s'_j \\ \mathbf{s}'' = s''_1,\ldots,s''_{|\mathcal{C}|}\end{array}}{\langle(\Psi,\mathbf{s}), i.\mathit{name}\,(v_1,v_2,\ldots)\rangle \to \langle(\Psi',\mathbf{s}''),N\rangle}$$

Next, we specify the semantics of function calls, which apply to both calls from external accounts (see first line of TRANS rule) and calls from other contracts. A function call is triggered by providing a contract instance $i \in \mathcal{C}$ with a function name $\mathit{name} \in \mathbb{I}$ and a list of parameter values $v_1, v_2, \ldots$. The execution first checks if there exists a transition (i.e., function) t with name $t^{name} = \mathit{name}$, if the origin state t^{from} of this transition t is the current state s_i, and if the guard condition g_t evaluates to `true` with the execution state σ initialized using the parameter values $v_1, v_2, \ldots$. A normal execution passes the above tests and executes the action a_t of the transition, resulting in a new execution state σ' and keeping the execution status normal N. Finally, it sets the current state of the contract s''_i to the destination state t^{to} of the transition, and yields the new ledger, contract, and destroyed states Ψ', $\mathbf{s}''$, and κ' as well as normal execution status N. This normal execution is captured by the FUNC rule:

$$\text{FUNC}\ \frac{\begin{array}{c} s_1,\ldots,s_{|\mathcal{C}|} = \mathbf{s}, t\in T_i, \mathit{name} = t^{name}, \\ s_i = t^{from}, M = \mathit{Params}(t,v_1,v_2,\ldots), \\ \sigma = (\Psi, M, \mathbf{s}, \kappa) \\ \mathit{Eval}(\sigma, g_t) \to \langle\sigma, R[\mathtt{true}]\rangle \\ \langle(\sigma,N),a_t\rangle \to \langle(\sigma',N),\cdot\rangle, \\ \sigma' = (\Psi', M', \mathbf{s}', \kappa'), s'_1,\ldots,s'_{|\mathcal{C}|} = \mathbf{s}', \\ s''_i = t^{to},\quad s'_1,\ldots,s''_i,\ldots,s'_{|\mathcal{C}|} = \mathbf{s}'' \end{array}}{\langle(\Psi,\mathbf{s},\kappa), i.\mathit{name}\,(v_1,v_2,\ldots)\rangle \to \langle(\Psi',\mathbf{s}'',\kappa'),N\rangle}$$

c) Global Coordinator Component: To enforce the execution of a single transaction at a time [19], we include in our generated BIP models a component that represents a global coordinator. This component coordinates the system execution so that only one transaction can

```
1  function kill(address _to)
2     onlymanyowners(sha3(msg.data)) external {
3    selfdestruct(_to); }
```

Fig. 7. Solidity code of `kill` function

be executed at a time. The BIP model of the coordinator is shown in Figure 6. It comprises three transitions: `beginTransaction`, `finishTransaction`, and `rmKilledContracts`, which are exported as ports in the interface of the component. At the end of each transaction, the Coordinator checks the state of the contracts in the system and removes any contracts that were self-destructed during the transaction through the synchronized execution of the `rmKilledContracts` transition.

d) Augmentation: In Figure 6, we also present part of the augmented transition system of the `WalletLibrary` contract. In particular, Figure 6 shows the augmented transition system of the `kill` function (the corresponding Solidity code is shown in Figure 7). Our algorithm extends the algorithm used in VeriSolid [10] by taking into account contract interaction mechanisms. In particular, for each function it adds transitions `beginCL` and `beginLL`.

The connectors between the ports of the Coordinator and the ports of the `kill` function define synchronization of transitions, e.g., transition `beginTransaction` and `beginCL` must be executed simultaneously. Further, through this connector, data is exchanged between the two components. In particular, `beginCL` sends the unique `id` of the contract to the Coordinator, which stores it in a variable. Similarly, through the connector between ports `finishTransaction` and `selfdestruct`, the unique `Id` of the contract is also sent to the Coordinator. The synchronized execution of `finishTransaction` and `selfdestruct` is enabled only if the two `Ids` match (guard of `finishTransaction`). This restricts the execution of a single transaction at a time.

When another contract calls the `kill` function, the synchronization will be between the transition `beginLL` and the transition that invoked `kill` from the other contract, which will then have to wait in the same state until it can synchronize with `selfdestruct`. These synchronizations enable function calls to be nested to any depth, but ensure that the caller always waits for the callee to finish. Notice that for `beginLL`, the guard condition and revert transition are not present. The reason for this is that due to the restricted interactions (high-level function calls, custom delegation, and `transfer`), either all functions run to a normal stop or all functions revert in our system. Consequently, if a callee reverts (either due to the guard not being met or for some other reason), then all calls must be reverted, which is captured for verification by the revert option of the top level call (see, e.g., `beginCL`).

Note that the connectors of the BIP models are automatically generated using the information given through the corresponding SDD and by statically detecting function and delegation calls in the body of a Solidity functions.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

e) Delegation calls: Since we restrict interactions between contracts such that exceptions are always rethrown, we introduce a custom delegation statement. The syntax of our custom, high-level delegation statement is:

```
contract.delegate.function(arg1, arg2, ...);
```

where $contract$ is reference to another contract, $function$ is a function name, and $arg1, arg2, \ldots$ are arguments. We omit the semantics of evaluating this expression here due to space restrictions that can be found in [18].

Our code generator implements the expression as a simple wrapper around `delegatecall` that rethrows exceptions:

```
if (!address(contract).delegatecall(
abi.encodePacked(bytes4(keccak256("
 function(arg1type, arg2type, ...)")),
        arg1, arg2, ...)))revert();
```

f) Verification properties: As shown above, our framework provides a clear separation of concerns between contract behavior and interaction, which allows one to compositionally model and analyze systems of interacting smart contracts. Once the BIP models are generated in Step 3, the user may specify temporal logic properties in CTL to verify the system.

Even if the user does not specify any properties, our framework by default always checks for deadlock freedom. It is interesting to note that for the Parity Wallet contracts, we are able to detect the parity bug by only checking for deadlock freedom. In particular, the counterexample returned by the NuSMV model checker included the following execution trace (after executing `initMultiowned`): 1) the `kill` function of `WalletLibrary` is called during a transaction; 2) in the end of the transaction, WalletLibrary is destroyed (goes to the destroyed state); 3) a new transaction begins where a function of `Wallet` is called (in our trace this was `isOwner`) that uses `delegateCall` to `WalletLibrary`.

Our framework allows the specification of CTL properties that reference actions from different components. For instance, we next provide examples of liveness and safety properties that we verified:

- **WalletLibrary.destroy** will eventually happen after **Coordinator.beginTransaction** & **WalletLibrary.beginCL**.
- if **WalletLibrary.initWallet** happens, **WalletLibrary.addOwner** can happen only after **WalletLibrary.initmultiowned** happens.

VII. Vulnerability Analysis

Our model checking approach can handle both typical and atypical vulnerabilities by verifying all desired safety properties that are specified for a contract. In this section, we discuss examples of well known vulnerability types that can be prevented or detected using our framework.

1) Re-entrancy: A smart contract may be vulnerable to this pattern if there exists a function call to an external contract that can be used to re-enter the caller function, allowing the callee to exploit the transient state of the caller. Another variant of the same vulnerability, *cross-function re-entrancy*, may be observed in the case where two functions share the same state.

TABLE I
Summary of Vulnerabilities Verified

Vulnerability	Prevention / Verification Technique
Re-entrancy	Prevented by locking the contract using intermediate state
Exception Mishandling	Prevented by using high-level calls, `transfer`, and custom high-level delegation
External Contract Referencing	Prevented using automated deployment
Shortening of Address	Verified during deployment
Locked Ether	Verified during modeling
Unprotected Suicide	Verified during modeling

We do not allow re-entrancy by design for external function calls. In particular, after a transition begins but before the execution of the transition action, the contract changes its state to a temporary one. As a result, none of the functions may be called externally before the transition finishes (or reverts). One might question this design decision since re-entrancy is not always harmful. However, there is evidence that it can pose significant challenges for providing security. First, supporting re-entrancy substantially increases the complexity of verification. Our framework allows the verification of a broad range of properties within seconds, which is essential for iterative development. Second, re-entrancy often leads to vulnerabilities since it significantly complicates contract behavior. Hence, we believe that prohibiting re-entrancy is a small price to pay for security.

2) Mishandled Exceptions: With low- and high-level function calls (e.g., `call` and `transfer`), Solidity handles exceptions in two different ways: *(i)* the callee contract returns `false` to the caller contract, or *(ii)* the exception is propagated back to the caller and its function is reverted. In the first case, if a developer forgets to check the return value, exceptions can lead to an unintended state of the contract.

Our framework avoids this vulnerability by allowing only high-level function calls, `transfer`, and a customized high-level delegation mechanism, which is implemented as a wrapper around the low-level `delegatecall` and raises an exception upon failure.

3) External Contract Referencing: Contracts that are already deployed in the network can be referred to by other contracts using their addresses. In Solidity, any address can be cast as a contract irrespective of whether the code represents the intended contract type. As an adverse result, during deployment, a contract creator can provide address for an arbitrary code unknown to the users.

There are two recommended ways to prevent this malicious behaviour: *(i)* hardcoding any external contract address if known before deployment; *(ii)* usage of the `new` keyword to create contracts. Our framework automatically applies these prevention techniques during code generation (adding code for instantiating contracts and providing addresses) and during deployment (supplying correct addresses).

4) Shortening of Address: Addresses in Ethereum are 20 bytes long. Address shortening vulnerability can be exploited by forcing a transfer to an account by passing an address of shortened length. Consider an example where an address with

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

a trailing 0 was created and loaded with an off-line balance of 1000 tokens. Then, a transfer request is made from a wallet with $256,000$ tokens, but with the trailing 0 excluded from the receiver address in the transfer function. Since `address` parameter is expected to be of a specific length, the missing bytes are added from the $amount$ parameter which is of type `uint256` and has lots of leading zeros. This creates an underflow, because there are not enough bytes in `uint256` and zeros are added to the end of the $amount$ value, shifting it to 256 bits long.

All of the addresses created using our framework are verified to be 20 bytes in length. Even the ones that are passed along with the Solidity code are checked to make sure that the length is the same before the deployment.

5) Locked Ether: As discussed with our running example `ParityWallet`, some contracts allow users to deposit Ether into a contract but not allow withdrawing it under any condition. These contracts have been called *greedy contracts* [4]. In `ParityWallet`, the issue arose due to the use of a separate library for withdrawing funds. Once the library was killed, there was no way for the wallet contract to release the deposits, thus becoming greedy. A recommended prevention technique for this vulnerability is to provide critical functions within the contract instead of resorting to an external library. Our framework can verify that appropriate withdrawal functions always remain reachable.

6) Unprotected Suicide: This is another vulnerability observed during the first attack of `Parity Wallet` as discussed in Section VI. Contracts can be deleted by using `selfdestruct` instruction either by a direct call to its code or using `delegatecall`. This will remove the code of the contract from the blockchain and the Ether left in the contract address is sent to a specified target. In cases where the authentication mechanism for a contract is inadequate, the `selfdestruct` instruction could be called by anyone on the contract. As exploited in the `Parity Wallet`, the contract was deleted using the `kill` function causing a deadlock and losing the ability to transfer or withdraw funds in wallets. Our framework can verify if a suicide statement can be reached using an unintended execution trace.

VIII. Related Work

Smart contract verification has been recently the focus of a lot of research. Various methodologies have been proposed catering to different vulnerabilities. Traditional symbolic execution techniques have been used in [3], [7], [20], [21], [22] by compiling smart contract source code to bytecode and representing bytecode in the format required by these tools for analysis of known/typical vulnerabilities. Tools like Securify [7] and Slither [23] fall under this category and they verify contracts by traversing through the code data-flow.

Further, there are tools that specialize in detecting a specific type of vulnerability. As an example, VERISMART [8], SMTCHECKER [24], Zeus [25] and Osiris [26] are tools used to detect integer over/underflows and division-by-zero paths and Sereum [27] is used to look for reentrancy vulnerabilities. Although targeting minimal set of vulnerabilities, these tools guarantee high precision compared to their predecessors.

Formal verification has also been applied in the field of smart contract analysis to check program correctness through rigorous mathematical models. Hirai [28] proposed formal verification using Ethereum bytecode. Bhargavan et al. [6] proposed a framework that translates EVM bytecode to F^* and verifies contract safety and correctness. Finally, Atzei et al. [29] formaly proved properties of the Bitcoin blockchain.

By creating semantics for a virtual machine, a one-time task depending on the network, and by providing the specification for a contract, the bytecode of any smart contract can be verified during runtime. A drawback of this approach is that it requires tedious manual processing. KEVM [30] formalized EVM semantics into the K-framework. Sereum [27] is a runtime verification tool, which uses taint analysis and checks for storage and control flow in the contract. Microsoft has recently added formal verification semantics explicitly for its Azure Blockchain Workbench as well as a built-in verifier VERISOL which uses state transitions and model checking to analyze contracts [31]. SmartDEMAP [32] is another deployment and management platform that comes with built-in tools for formal verification. A custom programming language is used to specify the safety properties of a smart contract.

Recently, there have been proposals for visual programming languages. These are basically design oriented languages that automatically generate the underlying smart contract code based on the specific structure and flow that is presented. The main objective behind this effort is to provide a clear understanding to the developer on the "interactivity" between components of the code. Babbage [33] was designed to express smart contracts in terms of mechanical components. Bamboo [34], Obsidian [35] and Simplicity [36] are other languages which specify contracts as state machine functions.

In comparison, the main advantage of our approach is that it allows developers to specify desired properties for both standalone and interacting smart contracts. Developers can use high-level model form to specify the properties instead of using low-level representation, e.g., `EVM bytecode`. In addition, we synchronize verification and deployment as a common framework allowing a contract to be published on a blockchain network once verified.

IX. Conclusion

We present an end-to-end framework that allows the verification, generation, and deployment of correct-by-design interacting Solidity contracts based on VeriSolid. This framework provides a clear separation between contract behavior and interaction, which allows one to compositionally model and verify systems of multiple interacting contracts. To enhance usability and understandability, the proposed work provides easy-to-use graphical editors for the specification of high-level models that include transition systems and SDDs. Even though security incidents often exploit contract interaction as witnessed by the Parity Wallet hack and other well known attacks, prior work on smart contract verification, vulnerability discovery, and secure development typically considers only individual contracts. To the best of our knowledge, this is the first work that provides a systematized approach for designing and verifying systems of interacting contracts.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.

References

[1] K. Finley, "A $50 million hack just showed that the DAO was all too human," Wired https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/, June 2016.

[2] L. H. Newman, "Security news this week: $280m worth of Ethereum is trapped thanks to a dumb bug," WIRED, https://www.wired.com/story/280m-worth-of-ethereum-is-trapped-for-a-pretty-dumb-reason/, November 2017.

[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, October 2016, pp. 254–269.

[4] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[5] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "VULTRON: catching vulnerable smart contracts once and for all," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE Press, 2019, pp. 1–4.

[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Short paper: Formal verification of smart contracts," in *Proceedings of the 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in conjunction with ACM CCS 2016*, October 2016, pp. 91–96.

[7] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[8] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for Ethereum smart contracts," *arXiv preprint arXiv:1908.11227*, 2019.

[9] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," in *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC)*, February 2018.

[10] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," in *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security (FC)*, February 2019.

[11] K. Nelaturu, A. Mavridou, A. Veneris, and A. Laszka, "Open-source implementation of extended VeriSolid," https://github.com/smartcontractsfc/verifier, accessed on 12/19/2019.

[12] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[13] S. Bliudze, A. Cimatti, M. Jaber, S. Mover, M. Roveri, W. Saab, and Q. Wang, "Formal verification of infinite-state BIP models," in *Proceedings of the 13th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 2015, pp. 326–343.

[14] Etherscan, "Parity multisignature wallet source code," https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4, accessed on 9/24/2019.

[15] N. B. Said, T. Abdellatif, S. Bensalem, and M. Bozga, "Model-driven information flow security for component-based systems," in *From Programs to Systems. The Systems perspective in Computing*. Springer, 2014, pp. 1–20.

[16] A. Mavridou, S. Emmanouela, S. Bliudze, A. Ivanov, P. Katsaros, and J. Sifakis, "Architecture-based design: A satellite on-board software case study," in *Proceedings of the 13th International Conference on Formal Aspects of Component Software (FACS)*, October 2016, pp. 260–279.

[17] G. D. Plotkin, *A structural approach to operational semantics*. Computer Science Department, Aarhus University, Denmark, 1981.

[18] smartcontractsfc, "Formalisms report," https://github.com/smartcontractsfc/verifier/blob/master/Formalisms_Report.pdf, accessed on 12/19/2019.

[19] Solidity Documentation, "Blockchain basics," https://solidity.readthedocs.io/en/v0.5.3/introduction-to-smart-contracts.html?highlight=transaction#blockchain-basics, accessed on 9/24/2019.

[20] Trail of Bits, "Manticore: Symbolic execution for humans," https://github.com/trailofbits/manticore, October 2018.

[21] B. Mueller, "Smashing Ethereum smart contracts for fun and real profit," 9th Annual HITB Security Conference (HITBSecConf), 2018.

[22] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A framework for high-level analysis of Ethereum bytecode," in *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2018.

[23] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[24] L. Alt and C. Reitwiessner, "SMT-based verification of solidity smart contracts," in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 376–388.

[25] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the 2018 Network and Distributed Systems Security Symposium (NDSS)*, 2018.

[26] C. F. Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2018, pp. 664–676.

[27] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[28] Y. Hirai, "Formal verification of deed contract in Ethereum name service," https://yoichihirai.com/deed.pdf, November 2016.

[29] N. Atzei, M. Bartoletti, S. Lande, and R. Zunino, "A formal model of Bitcoin transactions," in *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC)*, 2018.

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

[31] S. K. Lahiri, S. Chen, Y. Wang, and I. Dillig, "Formal specification and verification of smart contracts for azure blockchain," *arXiv preprint arXiv:1812.08829*, 2018.

[32] M. Knecht and B. Stiller, "Smartdemap: A smart contract deployment and management platform," in *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer, 2017, pp. 159–164.

[33] Reitwiessner, C, "Babbage: a mechanical smart contract language," https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e, accessed on 9/21/2019.

[34] Y. Hirai, "Bamboo: an embryonic smart contract language," https://github.com/pirapira/bamboo, accessed on 9/25/2019.

[35] M. Coblenz, "Obsidian: a safer blockchain programming language," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 97–99.

[36] R. O'Connor, "Simplicity: A new language for blockchains," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. ACM, 2017, pp. 107–120.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on February 16,2021 at 11:28:25 UTC from IEEE Xplore. Restrictions apply.