Correct-by-Design Interacting Smart Contracts and a Systematic Approach for Verifying ERC20 and ERC721 Contracts with VeriSolid

Keerthi Nelaturu, Anastasia Mavridou, Emmanouela Stachtiari, Andreas Veneris, Aron Laszka

Abstract—Blockchain-based smart contracts enable the creation of decentralized applications, which often handle assets of considerable value. While the underlying platforms guarantee the correctness of smart-contract execution, they cannot ensure that the code of a contract is correct. Today, as evidenced by a number of recent security breaches, developers still have a hard time making contracts that work properly.Even though these incidents often exploit contract interaction, prior work on smart-contract verification, vulnerability discovery, and secure development typically considers only individual contracts in isolation. To address this gap, we introduce the VERISOLID framework for the formal verification of contracts that are specified using a abstract state machine based model with rigorous operational semantics. Our model-based approach allows developers to reason about and verify the behavior of a set of interacting contracts at a high level of abstraction. VERISOLID allows the generation of Solidity code that is functionally and behaviorally equivalent to verified models, which enables the creation of correct-by-design smart contracts. We additionally introduce a graphical notation (called *deployment diagrams*) for specifying possible interactions between contract types. Based on this notation, we present a framework for the automated verification, generation, and deployment of contracts that conform to a deployment diagram. To demonstrate the applicability of VERISOLID, we translate existing Ethereum Improvement Proposal (EIP) specifications to temporal properties for two of the most popular contract interfaces: ERC20 and ERC721. We also show you how to write code for the ERC20 and ERC721 interfaces in a way that is safe, and we do this by using VERISOLID. We evaluate our framework on 726 contracts that are currently deployed on the Ethereum blockchain, which include 267 ERC20 and 459 ERC721 contracts. Our experiments indicate that 18% of ERC20 contracts and 4% of ERC721 contracts fail to satisfy the EIP specifications.

Index Terms—Smart contracts, Verification, Solidity, EIP, ERC20, ERC721, Ethereum

1 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 the most widely used application of blockchains today is undoubtedly the remittance of cryptocurrencies (e.g., BTC on Bitcoin network), blockchains can also be used as trustworthy decentralized mediums for general-purpose computation in the form of smart contracts.

As with any software implementation, smart contracts may suffer from subtle and surprising bugs introduced 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. The most notable security incidents include the "DAO attack" [1] from 2016, the "Parity Wallet hack" [2] from 2017, and a vulnerability in random-number generation in Fomo3d and LastWinner [3]. The recent surge in Decentralized Finance (DeFi) applications, which stand at a total market-cap of $24 billion as of 2021, along with the rise in DeFi-specific attacks indicate the need for smart-contract verification frameworks. Motivated by such security issues and incidents, a number of tools have been proposed for vulnerability discovery using varied techniques 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 [1].

In this paper, we propose an end-to-end framework for the *correct-by-design development* and *deployment* of Solidity smart contracts on the Ethereum blockchain network [4]. Our work builds upon the VERISOLID open-source framework [5], which supports the correct-by-design development of stand-alone contracts. In particular, VERISOLID allows developers to graphically design a smart contract as an Abstract State Machine(ASM), perform model checking [6], and generate functionally equivalent Solidity code based on formally defined operational semantics.

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

- We propose a graphical notation, called Solidity Deployment Diagram, for specifying allowed interactions between smart contracts.
- We extend the VERISOLID operational semantics to formally capture interactions between smart contracts.
- We introduce an approach for the formal verification of a system of interacting smart contracts using our global coordinator component.
- Previous versions of VERISOLID supported only generation of individual smart contract code. We extend the tool to support the generation of multiple interacting

contracts.

- We provide a deployment framework for the generated smart contracts on the Ethereum blockchain. This will allow for automated deployment after the contracts are verified to be valid according to the specifications.
- We extend the framework to facilitate automated verification of contracts that are already deployed on the Ethereum blockchain. Thanks to this feature, we are able to process existing smart contracts as well in VERISOLID.
- We demonstrate the utility of the framework by translating existing Ethereum Improvement Proposal (EIP) specifications to temporal properties that are automatically applied during the verification of contracts that implement EIP interfaces.
- We classify the implementation paradigms of two EIP interfaces, ERC20 and ERC721, thereby providing a reference of safety guidelines when developing and deploying contracts that implement these interfaces.
- We provide experimental results for our framework through the verification of 726 contracts (267 ERC20 and 459 ERC721 contacts) that are currently deployed on the Ethereum blockchain and in use by various applications. Additionally, we study verification times using VERISOLID to relate reachable states in the contract with the time required to verify it.

The remainder of this paper is organized as follows. Section 2 describes our novel method for dealing with multiple interacting contracts. Section 3 we introduce VERISOLID framework and discuss concepts extended in this work for modeling interactions in smart contracts. Section 4 we define the formal semantics for all of the modelling concepts discussed in the previous section. In Section 5 we present our verification methodology for both standalone and interacting smart contracts. Section 6 we shift the focus on specific applications of smart contracts, i.e.,`tokens`. We introduce the topic of EIPs and define all the temporal properties that have been identified from EIP specifications of ERC20 and ERC721 interfaces. Section 7 describes various methodologies and safety guidelines that can be used while implementing ERC20 and ERC721 contracts. In Section 8 we briefly give an overview of the transition system generation and deployment diagram from an existing Solidity contract using ProxyERC20 contract as an example. Section 9 presents the evaluation of VeriSolid framework by conducting experiments on currently deployed contracts in Ethereum blockchain. Section 10 discusses related work and Section 11 provides the conclusion and future directions for our work.

2 AUTOMATED VERIFICATION 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 (https://github.com/smartcontractsfc/verifier) which allows the collaborative development of Ethereum contracts with built-in version control, which enables branching, merging, and history viewing. This repository is a fork of the initial VERISOLID implementation and has been modified to support the features added in this extension. We also extend it to support Solidity v0.5.

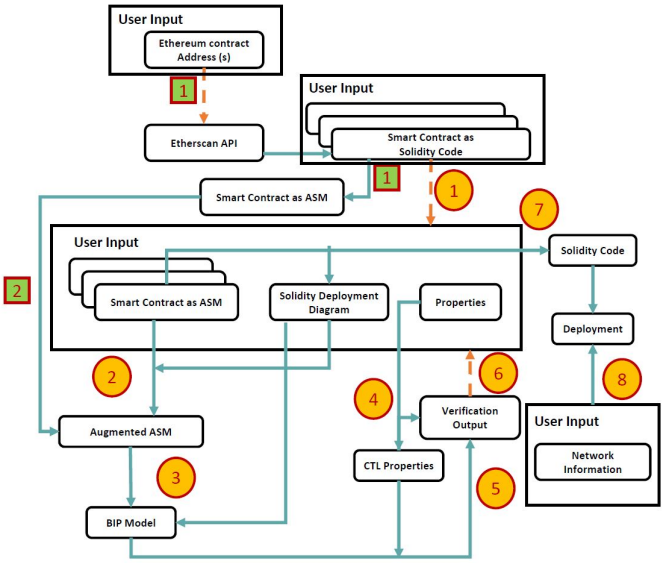

Fig. 1. Design, verification, and deployment workflow

Figure 1 shows the steps (numbered arrows) of the design, verification, and deployment workflow of our approach that extends the original VERISOLID framework. Squared numbers represent the one-click automated verification workflow which is an addition over the previous version of VERISOLID. Circled numbers represent steps that exist 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 ASM 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 ASM, as detailed in Section 8.
- 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 6).
- 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 3.2).
- Ethereum contract addresses for a contract. Providing the deployed contract address triggers the automated verification which will download the contract using an `API` provided by Etherscan – a blockchain explorer [7]. Depending on the contract type if they are implementing either ERC20 or ERC721 interface, the contract will be verified using the respective CTL (Computation Tree Logic) [8] properties that are pre-defined in our framework. We have provided background for CTL in our appendix for further reference [9]. Further to these

properties, user can provide additional specifications customized to their Decentralized Application (Dapp) using our existing workflow.

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 ASMs and the deployment diagram information. An augmented model of the contract is the extension of the initial ASM, where each original transition is replaced by a series of transitions, which have only simple Solidity statements as actions, and intermediate states that model the control flow of the Solidity-code action of the original transition. 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 CTL properties. The model can then be verified for deadlock freeness or other properties using tools from the BIP tool-chain [10] or `nuXmv` [11] (Step 5). If the required properties are not satisfied by the model (depending on the output of the verification tools) the input can be fixed 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. We discuss the representation of smart contracts as ASMs and SDDs with an example in Section 8. The scope of supported operations are listed in the appendix [9].

3 Modelling Smart contracts

3.1 VeriSolid Framework

VeriSolid enables developers to *(i)* specify smart contracts as Abstract State Machines (ASM), *(ii)* verify these systems individually, and *(iii)* generate functionally equivalent Solidity code from them [5]. To represent complex, dynamic systems, abstract state machines, previously known as Evolving Algebras, combine the declarative principles of first-order logic with the operational perspective of transitions [12]. As a result of the machine model's semantic definition, it is possible to deal with concurrent and reactive behavior in a direct and intuitive way; the fact that ASM-based system models naturally enable operational interpretations is often considered an advantage when dealing with complex technical systems. In our case, the states of an ASM 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 [5].

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 S. Further, let T denote the set of Solidity data types, I denote the set of valid Solidity identifiers, D denote the set of Solidity events and custom data type definitions, E denote the set of Solidity expressions, and C (E) denote the set of Solidity expressions that do not have any side effects (apart from consuming gas or raising an exception).

Definition 1. *An ASM for modeling a smart contract is a tuple* $(D; S; S_F; s_0; a_0; a_F; V; T)$*:*

- D D *is a set of custom event and type definitions;*
- S I *is a finite set of states;*
- S_F S *is a set of final states;*
- s_0 2 S, a_0 2 S *are the initial state and action;*
- a_F 2 S *is the fallback action;*
- V I T *contract variables (i.e., variable names and types);*
- T I S $2^{\mathsf{I}\times\mathsf{T}}$ C (T [;) S S *is a transition relation, where each transition* t 2 T *includes: transition name* t^{name} 2 I*; source state* t^{from} 2 S*; parameter variables (i.e., arguments)* t^{input} I T*; transition guard* g_t 2 C*; return type* t^{output} 2 (T [;)*; action* a_t 2 S*; destination state* t^{to} 2 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 2 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). In VeriSolid, formal verification is essential to check the behavioral correctness of the system under design. Other 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. In our case, since we have finite models for contracts, formal verification (e.g., by model checking) guarantees full coverage of execution paths for all possible inputs. We refer to our approach as *correct-by-design* due to the fact that, we take an ASM as an input and generate the NuSMV [11] transition system from the BIP system [13]. Using the semantics for Solidity we generate the functionally equivalent Solidity code from the ASM.

To illustrate how to represent smart contracts as ASMs, we use the *Blind Auction* example from prior work [14], which is based on an example from the Solidity documentation [15].

In a blind auction, each bidder first makes a deposit and submits a blinded bid, which is a hash of its actual bid, and then reveals its actual bid after all bidders have committed to their bids. After revealing, each bid is considered valid if it is higher than the accompanying deposit, and the bidder with the highest valid bid is declared winner. A blind auction contract has four main states:

1) `AcceptingBlindedBids`: bidders submit blinded bids and make deposits;
2) `RevealingBids`: bidders reveal their actual bids by submitting them to the contract, and the contract checks for each bid that its hash is equal to the blinded bid and that it is less than or equal to the deposit made earlier;
3) `Finished`: winning bidder (i.e., the bidder with the highest valid bid) withdraws the difference between her deposit and her bid; other bidders withdraw their entire deposits;

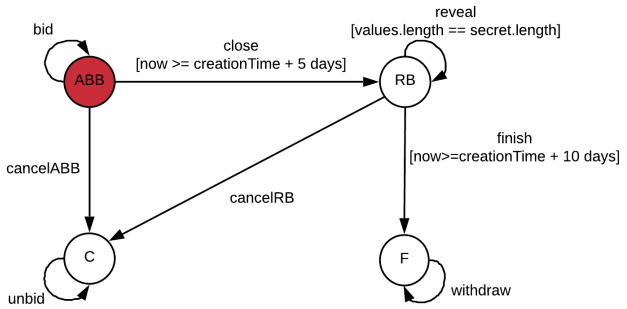

Fig. 2. Blind auction example as an ASM.

4) `Canceled`: all bidders withdraw their deposits (without declaring a winner).

This example illustrates that smart contracts have *states* (e.g., `Finished`). Further, contracts provide functions, which allow other entities (e.g., users or contracts) to invoke *actions* and change the states of the contracts. Hence, we can represent a smart contract as an *ASM* [16], which comprises a set of states and a set of transitions between those states.

Figure 2 shows the blind auction example in the form of an ASM. For ease of presentation, we abbreviate `AcceptingBlindedBids`, `RevealingBids`, `Finished`, and `Canceled` to `ABB`, `RB`, `F`, and `C`, respectively. The initial state of the ASM is `ABB`. To differentiate between transition names and guards, we use square brackets for the latter. Each transition (e.g., `close`, `withdraw`) corresponds to an action that a user may perform during the auction. For example, a bidding user may execute transition `reveal` in state `RB` to reveal its blinded bid.

3.2 Solidity Deployment Diagram

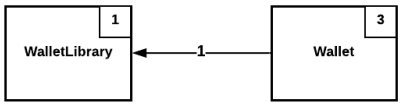

Fig. 3. SDD of Parity Wallet.

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*.

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. Figure 3 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.

Next, we present the necessary modeling concepts of the BIP component framework [10].

3.3 Modeling Interactions with BIP

Systems are modeled in BIP [17], [18] by superposing the *Behavior, Interaction,* and *Priority* layers. The *Behavior* layer consists of a set of components represented by ASMs. Each component transition is labeled by a *port*. Ports form the interface of a component 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* [17] layer since we do not use it in our contract models.

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 ASMs and BIP model. To support the augmentation process, we have extended the set of supported statements S by including `selfdestruct(`*@expression*`);` and our custom, high-level delegation invocation *@expression*`.delegate.`*@identifier*`(`(*@expression*(`,` *@expression*)`)?)`.

The syntax of our custom, high-level delegation statement is:

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

where *contract* is a reference to another contract, *function* is a function name, and *arg1*; *arg2*; ::: are arguments. The semantics of evaluating this expression can be found in Appendix [9].

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();
```

4 Operational Semantics

We now cover the operational semantics for interacting smart contracts and formally define the aforementioned concepts on deployment diagram.

4.1 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 [19]. Next, we present rules of the normal execution of a transaction and a function. These two rules, transaction and function call, are critical for the extended version of this work. The rationale for this is that these rules, when combined with SDD semantics, facilitate the implementation of interacting smart contracts. As a matter of readability, we have included all additional rules that capture both normal execution and exceptions for call delegation and nested function calls in Appendix [9]. We let denote the global 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 $\mathbf{s}$ denote the current state of contracts of the system. During the execution of a function, the execution state $= (; M; \mathbf{s};)$ also includes the memory and stack state M, and the set of destroyed contracts . 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 Eval(; Exp) $!$ $h\hat{}; R[v]i$ signify that the evaluation of a Solidity expression Exp in execution state yields value v with the new state $\hat{}$. On the other hand, Eval(; Exp) $!$ $h\hat{}; Ei$ signifies that the evaluation has resulted in an exception.

In our model, an externally owned account initiates a transaction by providing a contract instance $i \in C$ where C is a set of contract instances, with a function name $name \in \mathsf{I}$ and a list of parameter values $v_1; v_2; :::$. The transaction invokes the function in the current ledger and contract states and $\mathbf{s}$, which results in changed ledger and contract states $'$ and $\mathbf{s}'$. This normal execution without exception is captured by the TRANS rule:

$$\text{TRANS}\ \frac{\begin{array}{c} h(; \mathbf{s}; ;); i{:}name\,(v_1; v_2; :::)i \ ! \\ h('; \mathbf{s}';); xi; \quad x \in fN; R[v]g \\ 8j \in : s''_j = destroyed; \quad 8j \notin : s''_j = s'_j \\ \mathbf{s}'' = s''_1; :::; s''_{|\mathcal{C}|} \end{array}}{h(; \mathbf{s}); i{:}name\,(v_1; v_2; :::)i \ ! \ h('; \mathbf{s}''); N i}$$

According to this rule, a transaction is invoked by sending contract instance i, a function name and argument values $(v_1; v_2; :::)$. The call to the function is made on the current global state and the current contract states $\mathbf{s}$. If more contracts are invoked during the function execution, the state of those contracts represented by s'_j is updated. Any contracts that are destroyed during the function's execution are placed in k with their state s'_j set to *destroyed*. This way, we can keep track of the self-destructed contract states. If a return value v is specified at the end of the function, it is stored in the variable x as $R[v]$. If no return value is specified, the execution status N is returned instead. Finally, in $\mathbf{s}''$, all update states for contract instances that were invoked during the function execution are set to permanent. Additionally, the ultimate global state $'$ is permanently changed. $(; \mathbf{s}); i{:}name\,(v_1; v_2; :::)$ denotes the initial state when the transaction is initiated with the function call to `name`. $h('; \mathbf{s}''); N i$ represents the updated global and contract states after the execution of the transaction with the execution status N.

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 $i \in C$ with a function name $name \in \mathsf{I}$ and a list of parameter values $v_1; v_2; :::$. When a function call is made, the first thing to do is to check for the existence of the transition to the function in the model from the current state.

This normal execution is captured by the FUNC rule:

$$\text{FUNC}\ \frac{\begin{array}{c} s_1; :::; s_{|\mathcal{C}|} = \mathbf{s}; t \in T_i; name = t^{name}; \\ s_i = t^{from}; M = Params(t; v_1; v_2; :::); \\ = (; M; \mathbf{s};) \\ Eval(; g_t) \ ! \ h ; R[\texttt{true}]i \\ h(; N); u_t i \ ! \ h('; N); i; \\ ' = ('; M'; \mathbf{s}'; '); s'_1; :::; s'_{|\mathcal{C}|} = \mathbf{s}'; \\ s''_i = t^{to}; \quad s'_1; :::; s''_i; :::; s'_{|\mathcal{C}|} = \mathbf{s}'' \end{array}}{h(; \mathbf{s};); i{:}name\,(v_1; v_2; :::)i \ ! \ h('; \mathbf{s}''; '); N i}$$

When there is a transition $t \in T_i$ with the $t^{name} = name$ and the source state t^{from} is equal to the current state s_i, rule FUNC is applied. M must be provided with the parameter values $v_1; v_2; :::$ and the name of the transition t. The initial execution state will contain all of the aforementioned information, as well as the global state , all contract instance states $\mathbf{s}$, and a data structure k for storing modifications to destroyed contract states made throughout the function execution. If there is a guard condition g_t, as previously indicated, $Eval(; g_t)$ checks to see if g_t evaluates to $true$ without throwing an exception, which is represented by the return value $R[true]$. If g_t is *true*, then the action statement u_t is executed, which updates the final state to $'$ (see statement rules in the appendix [9]). This will include all contract states in $\mathbf{s}'$ being updated. If g_t evaluates to *false*, the action model's execution will be retracted using the revert transition that is included by default as part of model (see revert rules in appendix [9]). Finally, we set the current state of the contract s''_i to the destination state t^{to} of the transition, and yield the new ledger, contract, and destroyed states $'$, $\mathbf{s}''$, and $'$ when the execution status N is successful.

4.2 Semantics of Solidity Deployment Diagram

Definition 2. *An* SDD hT, n, Ai *consists of a set of contract types* $T = fT_1; :::; T_K g$ *where* $K \in \mathsf{N}$*; an associated cardinality function* $n : T \ ! \ \mathsf{N}$*, where* 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* $A = fA_1; :::; A_l g$ *of the form* $A = (ha_s; a_t i; d)$*, where* $a_s; a_t \in T$ *and* $l \in \mathsf{N}$ *are respectively the source and target of the deployment association, and* $d \in \mathsf{N}_{>0}$ *is the degree of the association.*

A deployment diagram is composed of the following components: A collection of contract types marked by $T = fT_1; :::; T_K g$. A contract's cardinality ($n(T_i)$), which specifies the number of instances that will be deployed for the contract. This can be any natural number. 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 noti es 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).

Fig. 4. Example deployment of Parity Wallet.

Next, we formally de ne a deployment instance (Definition 3) 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 4 conforms to the SDD of Figure 3.

De nition 3. A deployment is a pair $\langle \mathbb{C}, \iota \rangle$, where $\mathbb{C}$ is a set of contract instances and is a deployment con guration, i.e., a set of binary associations among the contract instances in $\mathbb{C}$. Each contract instance $C \in \mathbb{C}$ is speci ed as a pair $\langle T, v \rangle$ of contract type $T \in \mathbb{T}$ and constructor parameter values v, i.e., C is instantiated from type T with parameters v.

When we deploy a contract on Ethereum, we must also give any constructor arguments necessary to establish the initial contract state. This is accomplished by De nition 3. Speci cally, we evaluate all contract types, namely $\mathbb{C}$, and associate each contract type with its parameter values in ι. The developer supplies the values as part of the deployment diagram. The concept of deployment con guration is a crucial factor in order to deploy valid contracts. In particular when the constructor is expecting arguments but none are provided, Ethereum's deploy transaction raises an exception.

De nition 4. [Deployment Semantics] A deployment $\langle \mathbb{C}, \iota \rangle$ conforms to an SDD $\langle \mathbb{T}, n, \mathbb{A} \rangle$ if 1) for each $i \in [1; K]$ where K identi es the contract type in $\mathbb{T}$, the number of contract instances of type C_i in $\mathbb{C}$ is equal to n_i and 2) for each association $A = (\langle a_s; a_t \rangle, d) \in \mathbb{A}$ and instance $C_i \in \mathbb{C}$ such that C_i is of type a_s, there exist exactly d instances $C_j \in \mathbb{C}$ of type a_t such that $(C_i; C_j) \in \iota$.

Fig. 5. WalletLibrary : global coordinator and augmented component part.

The second condition can be written formally as follows:

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

Once the deployment diagram is created, and the deployment process is performed in VERISOLID, deployment semantics assist in ensuring that the diagram conforms to the permissible structure for the deploy transaction. Two checks are made in particular:

We determine whether or not each contract is associated to the correct number of instances, i.e., cardinality, to be deployed;

For each association, we validate its legitimacy and that the related contract has already been added to the deployment con guration;

5 VERIFYING INTERACTING SMART CONTRACTS

5.1 Global Coordinator Component

To enforce the execution of a single transaction at a time [20], 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 be executed at a time. The BIP model of the coordinator is shown in Figure 5. 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.

In Figure 5, we present part of the augmented ASM of the WalletLibrary contract. In particular, Figure 5 shows the augmented ASM of the kill function (the corresponding Solidity code is shown in Figure 6). We transform it to a functionally equivalent model that can be input into our veri cation tools. We perform three transformations: First, we replace the initial action a_0 and the fallback action a_F with transitions. Second, we replace transitions that have complex statements as actions with a series of transitions

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

Fig. 6. Solidity code of kill function.

that have only simple statements (i.e., variable declaration and expression statements). Third, we add the synchronization transitions that are to be executed along with the coordinator component. After these three transformations, the entire behavior of the contract is captured using only transitions. The transformation algorithms along with the observational equivalence proofs are discussed in detail in the appendix [9]. The input of the transformation is a smart contract de ned as a transition system (see De nition 1). The output of the transformation is an augmented smart contract

De nition 5. An augmented contract is a tuple $(D; S; S_F; s_0; V; T)$, where

- D: D is a set of custom event and type de nitions;
- S: I is a nite set of states;
- S_F: S is a set of nal states;
- $S_d \in S_F$ is the destroyed state;
- $s_0 \in S$, is the initial state;
- V: I T are contract variables (i.e., variable names and types);
- T: I S $2^{I \ T}$ C (T [;) S S is a transition relation (i.e., transition name, source state, parameter variables, guard, return type, action, and destination state).
- beginCL: T, a transition that takes the contract id as the parameter;
- beginLL: T, a transition added specically for kill function call.

Our algorithm extends the algorithm used in VERISOLID [5] by taking into account contract interaction mechanisms. In particular, for each function it adds transitions beginCL(begin Coordinator Lock) and beginLL(begin Local Lock).

The connectors between the ports of the Coordinator and the ports of the kill function de ne 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 relation to the FUNC and TRANS rules discussed in Section 4, synchronizations aid in isolating the state updates in s''. Permitting the states to be marked as destroyed as and when we reach u_t that is the selfdestruct statement. 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 Id s 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. The main purpose of this transition is to force the execution to wait at the beginLL transition until it can synchronize with selfdestruct. 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 veri cation 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. We identify the contract id s required for synchronizations by comparing the association parameters a_s and a_t to the deployment con guration mapping. This occurs whenever we encounter a u_t which is a delegate or function call to another contract.

5.2 Veri cation 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 [21] 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 speci cation of CTL properties that reference actions from different components. For instance, we next provide examples of liveness and safety properties that we veri ed:

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

6 EIP TRANSLATION TO TEMPORAL PROPERTIES

6.1 Introduction to EIPs

In this paper, additional to presenting the VERISOLID framework, we demonstrate the veri cation of Solidity contracts which implement EIP standard interfaces. EIP [22] is a design document that is used to de ne the architecture or describe the process and functionality behind any new

features solely designed for Ethereum. Our interest for this work is to analyze ERCs (Ethereum Request for Comment) which are application-level standard interfaces and conventions that are part of the EIP Standards Track. Particularly, we verify two of the most popular token standards: EIP-20 [23] and EIP-721 [24].

6.1.1 EIP-20: Token Standard (ERC20)

Tokens in general represent digital assets, such as IOUs which are informal documents specifying ownership or debt information, or even real-world, tangible objects. Tokens can be transferred or received between accounts. ERC20 de nes the standard APIs that are to be implemented when creating a token. The speci cation details the rules these APIs must adhere to. Some of these API methods include: transfer, transferFrom, approve, balanceOf and totalSupply . Using this standard enables integration of tokens with multiple wallets and exchanges seamlessly. As of March 2022, there were around 499,187 ERC20 tokens deployed on Ethereum.

6.1.2 EIP-721: Non-Fungible Token Standard (ERC721)

ERC721 is a token standard similar to ERC20 proposed in late 2017s, used for implementing Non-Fungible Tokens (NFTs). NFT refers to assets that can be physical (ex: houses) or collectible items (ex: cards) or negative values assets (ex: loans, burdens). A major difference between ERC20 and ERC721 is Fungibility. ERC20 token is fungible which allows it to be exchanged for another ERC20 token i.e., there is no distinction in terms of value between two tokens that belong to the same ERC20 contract. On the other hand, every ERC721 token has a unique value of its own. ERC721 includes all of the ERC20 methods and some additional methods which include: setApprovalForAll, ownerOf, safeTransfer and safeTransferFrom . As of March 2022, there are around 49,626 ERC721 tokens deployed on Ethereum.

In this work, we identify CTL properties from the EIP speci cation for ERC20 and ERC721. As of 2021, these two speci cations represent a substantial part of the contracts. Hence, we choose to demonstrate the applicability of VERISOLID by verifying conformance to these speci cations. Table 1 lists all the new CTL templates that we have introduced in this work, in addition to previous work [5]. We use p, q, r , s, and t to denote the transitions or statements, i.e., hTransitions [Statementsi. Transitions is a subset of the transitions of the model (i.e., Transitions T). A statement from Statements is a speci c inner statement from the action of a speci c transition (i.e., Statements T S). These properties are automatically applied during the veri cation of contracts that implement the speci c standard interfaces. The templates are used to evaluate the contract's adherence to the EIP. Our approach to identifying these properties is to consolidate all statements that include the keywords: MUST, MUST NOT, SHOULD, and SHOULD NOT. These keywords, which are used in EIP based on RFC2119 [25] indicate the absolute requirements of the speci cation.

TABLE 1
EIP Property templates

Property ID	Template	CTL formula
1	p must happen between q and r	AG(p ^ : r ! : E[: q U r])
2	p can happen	EF[p]
3	p or q or r must happen between s and t	AG(p ^ : r ! : E[(: q _ : s _ : t) U r])
4	if p happens, q must happen only after r happens	AG(p ! AX A[: q W r])

TABLE 2
ERC20 Absolute Requirements

ID	Rules from Speci cation	Property Used
1	Transfer SHOULD throw if the message caller's account balance does not have enough tokens to spend	
2	Transfer MUST re the Transfer event	
3	TransferFrom SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism.	Property 1
4	TransferFrom MUST re the Transfer event	
5	Approve MUST re the Approve event	

6.2 ERC20 Templates and CTL formulas

All of the rules that are mandatory for ERC20 have been listed in Table 2. We introduced an additional template below that can be used to verify all the 5 properties of the ERC20 token standard:

> p must happen between q and r.

The above template can be formulated in CTL as AG(p ^ : r ! : E[: q U r]) which expresses that globally we are looking for a path that starts with a state where p is true and r is false . This state is eventually followed by a state at which r is true if and only if there exists a path where q is true .

For example, Property 1 in Table 2 checks for safety property in the Transfer function. This function is used to make a transfer of the tokens from msg.sender to the address provided as the _to parameter. Our safety property veri es if the below checks are being made in the function:

- Determine whether msg.sender has enough tokens in their balance .
- Determine whether there is a mechanism in place to throw an exception when there are insuf cient tokens in the balance .

Both the above checks must always happen when the Transfer function is called and before the end/return of Transfer function. In Fig 7, we show an implementation of the Transfer function that is valid and satis es Property 1, discussed above. The CTL formula used for this would verify if there is a require statement that checks for account balance before the return of the Transfer function.

Property 3 is a safety property on the TransferFrom function. This function allows for the transfer of tokens on behalf of another account. It is required to check if the account has approved the msg.sender to transfer the tokens. Properties 2, 4, and 5 are used to verify if the requirement for the events Transfer and Approve to be red when the respective functions are called is satis ed. We use the augmented model of the contracts in order to verify all of the above properties.

6.3 ERC721 Templates and CTL formulas

ERC721 and ERC20 as mentioned in Section 6.1 are both token standard speci cations. ERC721 is used to create

```
1 function transfer(address _to, uint256 _value) returns (
      bool success) {
2     require(balances[msg.sender] >= _value && balances[
          _to] + _value > balances[_to]);
3     balances[msg.sender] -= _value;
4     balances[_to] += _value;
5     emit Transfer(msg.sender, _to, _value);
6     return true;
7   }
```

Fig. 7. Solidity code of transfer function in ERC20 contract

NFT tokens for which implementation varies from ERC20 because of the differing token. Hence, EIP speci cation has some additional rules that are to be adhered for a safe operation of the contract. All together we have recognized 12 properties for this interface which are listed in Table 3.

Of these 12 properties, we have reused the template de ned in Section 6.2 for verifying Properties 2, 3, 4, 5, 7, 8 and 11. Properties 2, 3, and 4 verify if the events Transfer , Approval and ApprovalForAll are emitted when the respective functions are called. Property 5 stipulates that the safeTransferFrom function analogous to the transferFrom function in ERC20, throws an exception if the address in _from parameter is not the owner. Property 7 ensures that safeTransferFrom function will not allow transfers to a 0x0 address which makes the tokens invalid. Property 8 and 11 both check if the transaction is going through a transfer or approval then the _tokenId passed to the functions must be valid. All of the above properties can be checked using our newly added templates by processing the speci c functions' augmented model.

Property 1 is used to check for compliance of an ERC721 contract. An ERC721 contract is expected to implement ERC165 interface for it to be compliant with ERC721. The ERC165 interace refers to EIP-165 speci cation which expects the implementation of one method supportsInterface . To verify the conformity to the standard we added the template below which allows to check if there exists a transition to supportsInterface action in our initial model of the contract. The formal CTL property for the same can be represented as $EF(p)$. This property can be used to check for the existence of a transition from initial state to a state at which p is true .

p can happen.

For properties 6 and 10 we de ne a new template below. An an example, if we consider Property 6, it requires us to verify a safety property on safeTransferFrom function. The function must throw an exception if either of the three cases is nottrue i.e., if msg.sender :

is not the owner of the token or
is not an authorized operator or
is not an approved account for the NFT

p or q or r must happen between s and t.

The CTL property for this template can be formulated as $AG(p \wedge r \; !: \; E(: q \_ : s \_ : t) \; U \; r])$ which expresses that, starting with the state where p is true and r is false there exists at least one state where either q or s or t is true . This state is eventually followed by a state at which r is true .

TABLE 3
ERC721 Absolute Requirements

ID	Rules from Speci cation	Property Used
1	MUST implement the ERC165 interface	Property 2
2	Transfer event MUST be emitted when NFT is created or destroyed	Property 1
3	Approval MUST emit Approve event when there is a change in approved address: changed or reaf rmed	Property 1
4	ApprovalForAll MUST be emitted when owner is enabled or disabled to manage the NFTs	Property 1
5	safeTransferFrom SHOULD throw if _from is not an owner	Property 1
6	safeTransferFrom SHOULD throw if msg:sender is not the owner or authorized operator or approved for the NFT	Property 3
7	safeTransferFrom SHOULD throw if _to is a zero address	Property 1
8	safeTransferFrom SHOULD throw if _tokenId is not a valid NFT	Property 1
9	if _to is a smart contract, onERC721Received is called and SHOULD throw if return value is not onERC721Received(address, address,uint256,bytes)	Property 4
10	Approve SHOULD throw if msg:sender is not the owner or authorized operator or approved for the NFT	Property 3
11	getApproved SHOULD throw if the _tokenId is not a valid NFT	Property 1
12	Contract MUST allow multiple operators per owner	Manual

```
1 require(spender == owner || getApproved(tokenId) ==
      spender || isApprovedForAll(owner, spender);
```

Fig. 8. Solidity code of require statement in ERC721 contract

Suppose the conditions q, s and t are all in a single statement as in Fig 8 we can directly use the template de ned for the ERC20 properties in Section 6.2.

We have used SDD discussed in Section 3.2 to specify the interaction rules and verify Property 9. For this property, we need to check if there is a delegate call to onERC721Received function in IERC721TokenReceiver interface. This should happen only when the _to address in safeTransferFrom function refers to a smart contract and not an externally owned account. Within the safeTransferFrom function, we also need to verify if the function throws an exception, in case the return value does not match with the signature of the onERC721Received function. We created the SDD for an ERC721 contract that interacts with an implementation of the IERC721TokenReceiver interface. Using an existing template in VERISOLID shown below, we can check if the SDD has the association to a particular function.

if p happens, q must happen only after r happens.

For our property, this template will be formulated as:

if to.isContract() happens, return (retval== _ERC721_RECEIVED) must happen only after IERC721TokenReceiver.onERC721Received happens.

Finally, Property 12 needs us to check if there are multiple operators per owner i.e., if there is a mapping (a Solidity variable type) from owner to a mapping of operatorApprovals . In order to verify this property we performed syntactical check for the variable declaration statement in the contract.

7 Implementation Paradigms for Standard Interfaces

As part of our experiments covered in Section 8, we have identi ed a few coding practices that have been used for both EIP-20 and EIP-721 standards. We were able to derive basic safety requirements for implementing the ERC20 and ERC721 interfaces by classifying these coding techniques. We determined the safety standards by manually analyzing the code of the various contracts used in the tests and labelling them according to the categories outlined later in this section. We identify these categories as safe, unsafe, and recommended approaches, which are de ned below:

1) Safe: This approach is in accordance with the ERC speci cation for the contract. Using this will avoid any known vulnerabilities.
2) Unsafe: Following this approach would make the smart contract unsafe and will make it vulnerable to attacks. Also, this is not in accordance with the ERC speci cation.
3) Recommended: By using VERISOLID, we have veri ed that this approach satis es the ERC speci cation and also adds additional checks to harden the smart contract. It is our recommendation that the contracts use this approach for implementation.

Our objective in developing these recommendations is to reduce attacks while adhering to all applicable regulations. As a result of the diverse interface structures that might be employed in conjunction with the standards, we classify contracts at the function and contract levels. All contracts utilized in the experiments were checked using VERISOLID to ascertain and con rm the strategy under which they may be classi ed.

7.1 Function-level Implementation Paradigms

ERC20 and ERC721, the standards we looked at in this study, include several safety features that are based on ensuring that there are checks for balances in place. Transfer , TransferFrom and Approval are the primary operations of these interfaces. We have identi ed eight paradigms based on the various implementations of these functions, as shown in Table 4. Of these, we recommend two types, which are safe and have the least possibility for vulnerabilities. Types 1F and 6F are considered safe if additional checks are added for over ows as discussed in this section. The others might get exposed to either integer over ows [26] or Fake deposit attacks [27] described below.

```
1  using SafeMath for uint256;
2  function transfer(address _to, uint256 _value) returns
       (bool success) {
3      balances[msg.sender] = balances[msg.sender].sub(
           _value);
4      balances[_to] = balances[_to].add(_value);
5      emit Transfer(msg.sender, _to, _value);
6      return true;
7  }
```

Fig. 9. Example transfer function using Type F1 implementation

```
1  function transfer(address _to, uint256 _value) returns
       (bool success) {
2      require(balances[msg.sender] >= _value &&
           balances[_to] + _value > balances[_to]);
3      balances[msg.sender] = balances[msg.sender].sub(
           _value);
4      balances[_to] = balances[_to].add(_value);
5      emit Transfer(msg.sender, _to, _value);
6      return true;
7  }
```

Fig. 10. Example transfer function using Type F2 implementation

Integer Over ow: Fixed-size integers up to a maximum of 256 bits are supported by Solidity. Integer over ow happens when any arithmetic operations produce numbers that are outside the range of a given integer.

Fake Deposit Attack: A Fake Deposit Attack has been detected majorly in ERC20 tokens with incorrect and improper implementation of the standard. The primary cause of this attack is because of the gap between the EVM behaviour and the developers' understanding of the semantics.

We refer to Properties 1 and 3 in Table 2. The expected effect of this property is to throw an exception whenever there is a violation in terms of the balance of msg.sender or authorization of _from account. In some implementations, instead of using throw , the code handles these violations by using the return statement. Unless otherwise handled by the user interface or the contract that is calling these methods, they would not revert the transactions. At least 4% of the existing ERC20 tokens are exposed to this vulnerability. Now we will discuss the categories in detail, using the transfer function in Figure 7 as our running example.

7.1.1 Type F1: Using SafeMath Library

One way to avoid integer over ows is to do the arithmetic computation, check the result, and rollback the transaction in the event of an over ow. SafeMath [28], a library developed by the Open Zeppelin team, contains intrinsic checks for all arithmetic functions. Developers may be certain that the necessary checks are being performed by importing this library and utilizing it for all arithmetic operations. As demonstrated in Figure 9, all arithmetic operations are performed using SafeMath . There are no require statements that check for over ows before making the calls to SafeMath .

7.1.2 Type F2: Using SafeMath Library along with require checks

TABLE 4
Function-level Paradigm Classi cation

Type	Description	Safe/Unsafe/ Recommended	Possible Vulnerability
F1	Using SafeMath library	Safe	Integer over ow
F2	Using SafeMath library and require checks	Recommended	
F3	Using SafeMath library and conditional check with return	Unsafe	Fake deposit attack
F4	Only require check	Unsafe	Integer over ow
F5	Using conditional check with throw	Safe	Integer over ow
F6	Using SafeMath library, conditional check with throw and require	Recommended	
F7	Conditional check with return	Unsafe	Fake deposit attack
F8	Proxy pattern with delegatecall	Safe	Malicious logic contract

```
function transfer(address _to, uint256 _value) returns
        (bool success) {
    if (balances[msg.sender] < _value &&
    balances[_to] + _value > balances[_to]) return false
        ;
    balances[msg.sender] = balances[msg.sender].sub(
        _value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
    return true;
    }
```

Fig. 11. Example transfer function using Type F3 implementation

```
function transfer(address _to, uint256 _value) returns
        (bool success) {
    if (balances[msg.sender] < _value &&
    balances[_to] + _value > balances[_to]) throw;

    balances[msg.sender] = balances[msg.sender].sub(
        _value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
    return true;
    }
```

Fig. 12. Example transfer function using Type F5 implementation

This type of implementation uses SafeMath in addition to require statements to check the variables for over ows. Type F1s' use of SafeMath alone has two drawbacks:

1) The contract may be rendered useless if a variable over ow occurs before the SafeMath functions are called, such as in a constructor .
2) There might be variables that do not use SafeMath and may cause over ow.

Checking for over ows may be done by placing a require statement in the constructor in case one. In the second scenario, regardless of how a variable is used, require checks must be performed on all variables. Type F2 transfer function is shown in Figure 10 as a sample implementation.

7.1.3 Type F3: Using SafeMath Library and conditional check with return

Despite the fact that SafeMath is utilized for arithmetic operations, the function will not throw an exception but simply return true or false in circumstances when other variables over ow. Figure 11 depicts a Type F3 instance. This is clearly a Fake deposit attack, as previously mentioned.

7.1.4 Type F4: Only require statements

SafeMath was not utilized at all in this type. Balances may only be checked using require statements. If the SafeMath function examines all variables, then this meets the EIP requirement. Otherwise, any uncontrolled arithmetic operations by the developer may result in integer over ows. Figure 7 depicts a running example of this type precisely.

7.1.5 Type F5: Using conditional check with throw

This type is quite close to Type F4. Rather than using require , the if control structure is used to validate the balances; if they are not valid, the control structure issues a throw statement. Even though it complies with the standard, this category provides the same danger as Type F4. Figure 12 illustrates an example of this kind.

7.1.6 Type F6: Using SafeMath Library, conditional check with throw and require statements

This category includes Types F1, F4, and F5. Three kinds of checks are introduced to assure the contract's safe execution. The SafeMath library is used for all arithmetic operations. Secondly, require statements are used to check for any additional variables. The if control structure validates the EIP standard checks and utilizes throw in the event of erroneous data or over ows.

7.1.7 Type F7: Conditional check with return

This is yet another glaring example of a Fake deposit attack and quite similar to Type F3. The difference being in this type, there is no SafeMath library used for arithmetic operations. By utilizing return rather than throw with if statement, the contract violates the EIP requirement. return , as explained in Type F3, will not revert transactions as expected unless handled by the user interface or the contract executing the function. This is an extremely risky method of developing token standards.

7.1.8 Type F8: Proxy pattern with delegatecall

The Proxy pattern is the nal category of function-speci c paradigms. Rather than implementing the methods within the contract, this type of pattern refers to another deployed ERC contract implementation on the network through its address. The methods are invoked using the delegatecall mechanism. This design was created to facilitate the issuance of upgradeable contracts. This will be discussed in further depth in Section 7.2.

7.2 Contract-level Coding Practices

In Section 7.1, we discuss categorization of standard contract implementation based on function-speci c details. This is essentially required to model the augmented ASM in VERISOLID. Over the last few years, there were new coding practices that have been created for EIP standard implementations. These practices are an improvement over the EIP standard speci cations. Table 5 lists these standards.

TABLE 5
Contract-level Coding standards

Type	Description
C1	Proxy pattern with delegate call
C2	ERC Upgrade-safe Implementation
C3	Using full implementation with metadata separated
C4	Using multiple tokens together
C5	Using basic without additional safety features
C6	Not adhering to the standard

7.2.1 Type C1: Proxy pattern with delegatecall

Due to the immutability of blockchains, a signi cant concern is that once launched, contracts cannot be updated at the same address location. Previously, a x was to re-deploy the contract with any modi cations to a new address location and then use the new address in all apps instead of the old one. This strategy is inconvenient and necessitates a user interface modi cation as well. Upgradeable solutions have been recommended to help streamline the upgrading process, particularly for EIP standards. These contracts are referred to as proxy pattern contractsThe logic of the contract is segregated into a separate contract in this design. The logic contract is deployed as a standalone contract, and the main contract makes calls to it using a delegatecall mechanism. Rather than utilizing the constructor to establish the main contract, we utilize a contract-level initialize public method. This method may be used at any point throughout the contract's lifespan to change the logic. A signi cant issue is that the user interface never directly invokes the logic contract; instead, it interacts with the main contract. This does not exclude a malicious actor from directly communicating with the logic contract. As a result, the developer must ensure that the initialize function and the logic contract's methods are only accessible to authorized users. For instance, this authorized user may be the contract's owner.

7.2.2 Type C2: Upgrade-safe implementation

This is comparable to Type C1 but is speci c to the implementations of the EIP standard. The upgradable main contract of the EIP standard contracts includes the SafeMath library. This is to guarantee that all necessary arithmetic checks are performed prior to invoking the logic contract, hence avoiding any exceptions.

7.2.3 Type C3: Using full implementation with metadata separated

In all the EIP token standard contracts, there are a few traditional methods: name, symbol and tokenURI , which are common to all standards. An interface called IERCMetadata is implemented instead of re-writing the logic for these functions for all the standards. This ensures that all the contracts using this interface can be called on these three methods. The other interface used is IEnumerable , which implements three functions: totalSupply , tokenOfOwnerByIndex , and tokenByIndex . This is another interface that guarantees that all methods that can enumerate through the tokens are implemented. With these interfaces, there is a separation and structure to the logic of the standard contracts that is achieved using which exchanges and user interfaces can have the common functions called without any issues. The ERCFull implementation uses these interfaces in addition to the standard ones.

7.2.4 Type C4: Using multiple token standard in a single contract

We have realized that there are certain ERC tokens which include both ERC-20 and ERC-721 interfaces i.e., the contract can support both Fungible and Non-fungible tokens. The implementation involves creating all the methods for both the standards with logic and storage separated. This is not a safe and suggested approach. Instead we suggest developers use EIP-1155 [29] which is a Multi-token standard that has speci cations to allow careful implementation of multiple tokens in a single contract.

7.2.5 Type C5: Using basic interface without additional safety features

Type C5 is a straight forward implementation of the EIP standard without any additional safety or upgrade features.

7.2.6 Type C6: Not adhering to the standard

Until recently, there were no EIP standards. Since the start of Ethereum blockchain there have been many tokens that were deployed without any standard speci cations being followed. These contracts come under this pattern. They all fail to comply with the EIP standards.

7.3 Discussion

We developed speci c safety rules for developers to follow when implementing token standards by evaluating these categories and coding approaches.

Rule (i) To maintain consistency with other token contracts of a similar nature, utilize a full implementation. This would make it easier for various exchanges and existing user interfaces to adopt your contract.

Rule (ii) Always use the SafeMath library for arithmetic operations.

Rule (iii) Use require statements to check whether there is room for any over ows with any of the variables.

Rule (iv) Make the contracts upgrade-safe with the logic of the contract separated.

Rule (v) A nal and an important rule is to run the contract through VERISOLID, which will verify for all these rules to be satis ed apart from the default checks for re-entrancy, deadlock freeness, and integer over ows. VERISOLID ensures that the contracts are correct-by-design and deploy them to the network automatically.

Fig. 13. ERC20 tokens categorized by function-speci c types.

Fig. 14. Veri cation results for ERC20 tokens.

In our experiments, we have used function-speci c categorization for ERC20 tokens and contract-level categorization for ERC721 tokens. Figure 13 shows the ERC20 tokens categorized by function-speci c types. 33% of the contracts use Type F2 or Type F6, which are the two recommended approaches based on the safety guidelines proposed for this work. Type F3 and Type F7, which are both direct representations of Fake deposit attacks cover 13% of the contracts. Figure 14 shows the veri cation results of the contracts. Most failures are observed in Type F3 and Type F7. 94 contracts have implemented the recommended safe types, i.e., Type F2 and only one contract of Type F1 was not complying with the EIP-20 standard. Type F6, which is also another recommended approach, had only 1 contract that passed all the properties.

Figure 15 shows the categorization of ERC721 tokens

Fig. 15. ERC721 tokens categorized by function-speci c types.

Fig. 16. Veri cation results for ERC721 tokens.

by contract-level coding practices. Of the 459 ERC721 contracts, 43% of them use ERCFull implementation, which is Type C3 under contract-level categories. This accounts for the majority of these contracts. These contracts are not upgrade–safe and thus do not comply with our policies. 22% of the contracts account for Type C5, which is basically implementing the EIP-721 standard without any additional safety features like SafeMath . Around 11% of contracts implemented multiple tokens without using the EIP-1155 standard. Although most of them do not fail in terms of EIP-20 and EIP-721 speci cation, using EIP-1155 is highly suggested for these contracts. In Figure 16, we demonstrate the veri cation results for ERC721 contracts. A total of 4% contracts failed to satisfy the properties of the EIP-721 standard. This is comparatively less when compared to the total failures observed in ERC20 contracts. This shows that the community is arriving at a maturity where the contracts are made sure to adhere to the EIP standards better than before, when initially only ERC20 tokens existed. 21% of the failed contracts implemented multiple tokens, and this brings in the cause to avoid using this type of implementation. It is suggested that all these contracts use EIP-1155 standard. 57% of the failed contracts do not have any safety features added, and they belong to Type C5 in the contract-level coding practices.

8 ASM Generation

In Step 1, a developer may import the Solidity code of an existing contract and use our automatic mechanism, algorithms for which are described in Appendix [9], to generate a corresponding ASM. After the extension, the generated transitions may contain guard conditions depending on whether the corresponding functions use Solidity modi ers. Note that not all modi ers can be converted into guards. The modi ers 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:

```
hguard_modifier i ::=
modifier @identifier ( @type @identifier
                       (, @type @identifier ) )
                       f (if ( @expression) ) j
                        (require ( @expression) ; )
                       j _; g
```

Fig. 17. ASM of ProxyContract .

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

As an example, we consider a variation of the Upgradeable ProxyERC20 contract implementation by Synthetix [30] which can be upgraded to a different target ERC20 contract. The implementation includes a set of two contracts: Proxy and ProxyERC20 . We automatically generated the ASMs of both the contracts by importing their source code, which is available on Etherscan [31] (280 lines of code for Proxy and 100 lines of code for ProxyERC20). Proxy is the user facing contract. Users can call the ERC20 functions in the target contract using Proxy . It can be done either using delegatecall or call mechanism.

From the source code of Proxy , our framework generated the model in Figure 17. The setTarget and setUseDELEGATECALL methods can be called by the owner only. Hence, the use of onlyOwner modifier. Using the setTarget owner can change the underlying ERC20 contract. The fallback function includes the actual code which makes the calls to the ERC20 contract. To verify that the source code and the ASM generated from it are equivalent, we re-generated the source codes of 500 contracts from the generated ASMs and compared the re-generated source codes to the original ones. We observed no changes to functionality, apart from adding statements that implement self-loop state transitions, which "lock" contracts. The impact of these "locks" is to prevent re-entrancy, which is a built-in security feature of VERISOLID . The only additional impact associated with this is the increased gas cost required to deploy the contract and to execute these functions owing to the newly added lines of code.

Fig. 18. ASM of ProxyERC20 .

Similarly, from the source code of ProxyERC20 , our framework generated the model shown in Figure 18. The initial transition ProxyERC20 corresponds to the constructor of the contract. To increase readability of the model, we have grouped all the methods into four transitions. Each generated ASM contains a single state. The Initialized state of Proxy represents the main state of the contract, which is entered once the owners are set up. We were able to test the model for the conformance with the EIP standard. The target ERC20 contract can be considered to fall under Type F2 which is one of the recommended approaches for ERC20 implementations.

Fig. 19. SDD of Proxy and ProxyERC20 .

Using the model generated, we regenerated the Solidity code and used SDD to deploy the contract in a local testnet. Figure 19 shows the SDD used for Proxy and ProxyERC20 contract.

9 EVALUATION

This section evaluates the efficacy and validity of EIP standard verification using VERISOLID. Additionally, we present the results of comparison of the reachable states and verification time for 35 contracts containing a range of contracts.

9.1 Dataset Collection and Environment

Our focus in this work was to verify contracts that use EIP-20 and EIP-721 standards. We collected 300 ERC20 contracts and 526 ERC721 contracts from Etherscan. Of these contracts, 4 ERC20 contracts and 67 ERC721 contracts consisted of only bytecode and no Solidity code. VERISOLID can be used to verify only Solidity code contracts. Hence, we excluded contracts with just bytecode and no Solidity code from our verification process. Within the 296 remaining ERC20 contracts, we observed that 29 contracts used a proxy pattern with delegate functionality. Contracts implementing this pattern refer to the EIP using proxy; all calls to the contract are delegated to the proxy. Since there is no direct access to the implementation, we excluded the proxy contracts as well. We verified 267 ERC20 tokens and 459 ERC721 tokens as part of our experiments. In our one-click automated verification, we just need to provide the list of addresses for currently deployed contracts on Ethereum that are to be verified. The framework will automatically recognize the contract standard type based on the interfaces being implemented. All the contracts have gone through manual inspection as well in order to identify any false positives. VERISOLID is a very light-weight application that can be used on a workstation with minimal requirements. The workstation we used for our experiments is equipped with Intel Core i3-4170 3.7GHz CPU, 16GB of DDR3 RAM, 512GB SSD running Linux Ubuntu 14.04LTS, in a local network environment.

Fig. 20. Veri cation results ERC20 tokens by Properties

Fig. 21. Veri cation results ERC721 tokens by Properties

9.2 Veri cation Results

Of the 267 ERC20 contracts, 49 contracts failed to con rm at least one of the rules we covered in Table 2. Figure 20 shows the variation in the types of failures witnessed in these contracts. Rules 2,4 and 5 require the speci cation of events and ring them at the respective locations. Ten contracts failed to conform to these rules. Decentralized applications with front-end are dependent on event logs to check that state has been updated. In cases where the events are missing, it is impossible to inform the user about the transactions' success or failure. Rules 1 and 3 refer to the balance checks before making a transfer from accounts. If failed to adhere to these checks, there are chances for integer over ows or excess of tokens being transferred or deducted from the funds. Forty-one of the contracts do not observe these checks, which is concerning.

459 ERC721 contracts were tested using VERISOLID. 19 contracts failed to adhere to the speci cation. That leaves 95% of contracts that have satis ed all the 12 rules mentioned in the EIP speci cation. The increase in the acceptance percentage can be related to the Ethereum community's maturity in understanding the importance of EIP speci cations. Most contracts are re-used for multiple applications depending on their availability on Etherscan and the open-source community. Figure 21 shows the distribution of the failures in terms of the 12 rules. Three of the failed contracts have not used the EIP-721 speci cation at all. These contracts are observed to be deployed in 2017, which is way before the speci cation was proposed in 2018. Eleven contracts failed to make the checks for balances similar to Rules 1 and 3 in ERC20.

Overall, 9% of the total contracts veri ed failed to conform to the speci cation. All of the contracts have been manually inspected to study for any false positives or false negatives. There were no false positives or false negatives observed in veri cation results.

Finally, Figure 22 shows the veri cation time of VERISOLID as a function of reachable states for the top 35 ERC20 tokens with the largest market capitalization according to Etherscan.

Fig. 22. Veri cation time and the number of reachable states for various contracts.

10 Related Work

Smart contract veri cation 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 [32], [33], [34], [35], [36] 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 [33], Ethainter [37] and Slither [38] fall under this category and they verify contracts by exploring through the code data- ow.

Furthermore, there are tools that specialize in detecting a speci c type of vulnerability. As an example, VERISMART [39], SMTCHECKER [40], Zeus [41] and Osiris [42] are tools used to detect integer over/under ows and division-by-zero paths and Sereum [43] is used to look for reentrancy vulnerabilities. Although targeting minimal set of vulnerabilities, these tools guarantee high precision compared to their predecessors.

Traditional fuzzing technique has also been in use for smart contract veri cation over the recent years. Fuzzing involves automated generation of inputs for testing based on the contract ABI speci cation which are passed to test oracles that determine vulnerabilities based on the results. ContractFuzzer [44] and sFUZZ [45] are two works that use fuzzing along side static analysis to identify known vulnerabilitis in a smart contract.

There has been a lot of work using Mutation Testing for veri cation [46], [47], [48]. This technique involves mutation of the contract code to either analyze using existing test cases for faults or to generate vulnerability-free contracts. Gas-aware and automated mutation testing has also been considered while searching for valid smart contract implementation with this approach.

Formal veri cation has also been applied in the eld of smart contract analysis to check program correctness through rigorous mathematical models. Hirai [49] proposed formal veri cation using Ethereum bytecode. Bhargavan et al. [50] proposed a framework that translates EVM bytecode to F and veri es contract safety and correctness. Finally,

Atzei et al. [51] 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 [52] formalized EVM semantics into the K-framework. Sereum [43] 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 [53]. SmartDEMAP [54] 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. sGUARD [55] focuses on transforming a smart contract with four known vulnerabilities automatically during runtime.

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. Babbage [56] was designed to express smart contracts in terms of mechanical components. Bamboo [57], Obsidian [58] and Simplicity [59] are other languages which specify contracts as state machine functions.

When it comes to conformance testing for tokens in particular, Tokenscope [60] is a project that is very similar to ours. Tokenscope is a tool that employs trace recording and monitoring in order to identify inconsistencies in ERC-20 contracts behavior. These anomalies are identified mostly by monitoring the traces for invalid token balance data structure updates. In contrast to their method, VERISOLID is not standard-specific. We have created the tool in such a manner that it can be used with any EIP standard to uncover known and unknown vulnerabilities in contracts depending on the specification. Another point of contention is Tokenscope's approach to targeted data structures. We are not interested in any specific data format, such as the token's balance. We develop a model of the contract and then seek for vulnerabilities based on all the state transitions and reachable states. Smart contract engineering (SCE) [61] is another effort that makes use of conformance testing to uncover errors in smart contracts. At the moment, SCE needs a significant amount of human effort to prepare the conformance test sets prior to doing the actual testing. A recent work on Wasm Smart contracts [62] also uses conformance checking to locate inconsistencies. But their tool is integrated with K framework [52] that was discussed previously.

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. Most of the verification frameworks try to focus on particular set of known vulnerabilities. We identify both typical and atypical vulnerabilities based on the specification provided. With our current extension to the work, we can verify existing deployed smart contracts on Ethereum with just one-click by providing the address of the contract. Specifically, in case of ERC20 and ERC721 contracts which are identified automatically, we check and provide feedback on the safety implementation practices and EIP specification. There is no necessity for the user to provide a specification unless it varies from the EIP specification.

11 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. The proposed work provides easy-to-use graphical editors for the specification of high-level models that include ASMs and SDDs. Specifically, in this work we target the demonstration of end-to-end verification from specification of EIP interfaces: ERC20 and ERC721. We also provide safety guidelines for implementation of these interfaces based on our classification of current implementation methodologies. To the best of our knowledge, this is the first work that provides a systematized approach for designing and verifying systems of interacting contracts.

With the growth of public blockchains over the last decade, languages used for smart contracts have proliferated. VERISOLID is presently limited to Solidity; however, the procedure is well-suited for automated deployment on the Ethereum blockchain. As a result, the tool is compatible with the Ethereum Virtual Machine (EVM) and Solidity-based blockchains. In future work, we will investigate the possibility of implementing an automated model-checking tool that might assist in testing properties in other smart contract languages. Our next target will be Move, the smart contract language of Diems [63], [64]. As another major direction, we will provide a tool that enables support for hyper-property verification in smart contracts. While model checking has been examined previously for the purpose of validating hyper-properties [65], [66], relatively little work has been done in the context of blockchains and smart contracts. In light of this, we believe this to be a worthwhile future direction for our work.

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] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, “Smart contract vulnerability analysis and security audit,” IEEE Network, vol. 34, no. 5, pp. 276–282, 2020.

[4] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” Ethereum Project – Yellow Paper, Tech. Rep. EIP-150, April 2014.

[5] 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.

[6] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, Sep. 1994.

[7] , "Etherscan," https://etherscan.io/, 2020, accessed on 12/10/2020.

[8] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*. Springer, 1981, pp. 52–71.

[9] K. Nelaturu, A. Mavridou, A. Veneris, and A. Laszka, "Appendix of correct-by-design interacting smart contracts and a systematic approach for verifying ERC20 and ERC721 contracts with VeriSolid," https://figshare.com/articles/online_resource/Verisolid_TDSC_appendix_pdf/18313640.

[10] 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.

[11] 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.

[12] R. e Karges, "Abstract state machine semantics of sdl," *Journal of Universal Computer Science*, vol. 3, no. 12, pp. 1382–1414, 1997.

[13] M. Noureddine, M. Jaber, S. Bliudze, and F. A. Zaraket, "Reduction and abstraction techniques for BIP," in *Proceedings of the International Workshop on Formal Aspects of Component Software*. Springer, 2014, pp. 288–305.

[14] 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.

[15] Solidity by Example, "Blind auction," https://solidity.readthedocs.io/en/develop/solidity-by-example.html#blind-auction, 2018, accessed on 9/25/2018.

[16] Solidity Documentation, "Common patterns," http://solidity.readthedocs.io/en/develop/common-patterns.html#state-machine, 2018, accessed on 9/25/2018.

[17] 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.

[18] 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.

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

[20] 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.

[21] NuSMV Model Checker, https://nusmv.fbk.eu/, 2022, accessed on 01/19/2022.

[22] Martin Becze, Hudson Jameson, "Ethereum improvement proposals," https://eips.ethereum.org/EIPS/eip-1, 2020, accessed on 12/10/2020.

[23] Fabian Vogelsteller, Vitalik Buterin, "Erc-20: Token standard," https://eips.ethereum.org/EIPS/eip-20, 2020, accessed on 12/10/2020.

[24] William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs, "Erc-721:non fungible token standard," https://eips.ethereum.org/EIPS/eip-721, 2020, accessed on 12/10/2020.

[25] Bradner, S, "Rfc2119," https://tools.ietf.org/html/rfc2119, 2020, accessed on 12/10/2020.

[26] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*. Association for Computing Machinery, 2020, p. 110–115. [Online]. Available: https://doi-org.myaccess.library.utoronto.ca/10.1145/3377644.3377650

[27] R. Ji, N. He, L. Wu, H. Wang, G. Bai, and Y. Guo, "Deposafe: Demystifying the fake deposit vulnerability in ethereum smart contracts," *arXiv preprint arXiv:2006.06419*, 2020.

[28] Open Zeppellin, "Safemath library," https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol, 2020, accessed on 12/10/2020.

[29] Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, Ronan Sandford, "Erc-1155 multi-token standard," https://eips.ethereum.org/EIPS/eip-1155, 2020, accessed on 12/10/2020.

[30] Synthetix, https://docs.synthetix.io/, 2022, accessed on 02/28/2022.

[31] ——, https://etherscan.io/address/0x23348160D7f5aca21195dF2b70f28Fce2B0be9fC#code, 2022, accessed on 02/28/2022.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities." in *PLDI*, 2020, pp. 454–469.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.

[45] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," *arXiv preprint arXiv:2004.08563*, 2020.

[46] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen, "Mutation testing for ethereum smart contract," *arXiv preprint arXiv:1908.03707*, 2019.

[47] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *arXiv preprint arXiv:1912.05823*, 2019.

[48] J. J. Honig, M. H. Everts, and M. Huisman, "Practical mutation testing for smart contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2019, pp. 289–303.

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

[50] 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.

[51] 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.

[52] 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.
[53] 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.
[54] 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.
[55] T. D. Nguyen, L. H. Pham, and J. Sun, "sguard: Towards fixing vulnerable smart contracts automatically," *arXiv preprint arXiv:2101.01917*, 2021.
[56] Reitwiessner, C, "Babbage: a mechanical smart contract language," https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e, accessed on 9/21/2019.
[57] Y. Hirai, "Bamboo: an embryonic smart contract language," https://github.com/pirapira/bamboo, accessed on 9/25/2019.
[58] 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.
[59] 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.
[60] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1503–1520.
[61] K. Hu, J. Zhu, Y. Ding, X. Bai, and J. Huang, "Smart contract engineering," *Electronics*, vol. 9, no. 12, p. 2042, 2020.
[62] R. Hjort, "Formally verifying webassembly with kwasm towards an automated prover for wasm smart contracts," 2020.
[63] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. R. Rain, S. Sezer *et al.*, "Move: A language with programmable resources," *Libra Assoc.*, 2019.
[64] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett, and D. L. Dill, "The move prover," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 137–150.
[65] B. Finkbeiner, C. Hahn, and H. Torfah, "Model checking quantitative hyperproperties," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 144–163.
[66] B. Finkbeiner, "Model checking algorithms for hyperproperties," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2021, pp. 3–16.

Keerthi Nelaturu is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering with focus on smart contract verification. Her research interests include decentralized blockchain oracles and smart contract verification/synthesis.

Anastasia Mavridou is a member of the Robust Software Engineering (RSE) Group at NASA Ames Research Center, employed by KBR Inc, where she performs research on formal methods and software engineering with a focus on requirements engineering, component-based system design, and compositional analysis techniques. Before joining RSE, she was a Postdoctoral Scholar at Vanderbilt University from 2017 to 2018. Anastasia received a Ph.D. degree in Computer Science from École Polytechnique Fédérale de Lausanne (EPFL) in 2016, an M.S. degree in Computer Science from the University of Tulsa in 2012, and a Diploma in Electrical and Computer Engineering from the Aristotle University of Thessaloniki in 2010.

Emmanouela Stachtiari has been post-doctoral researcher at the University of Geneva since 2020. Her research interests are model-based design and formal verification of systems and software. She received a Ph.D degree and an M.S. degree in Computer Science from the Aristotle University of Thessaloniki in 2018 and 2011, respectively, and a B.Sc degree in Applied Informatics from the University of Macedonia in 2009. Prior to her current role, she had worked in various R&D projects related to software engineering in academia and in industry.

Andreas Veneris is a Professor in Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto. His research interests include CAD for debugging, verification, synthesis and test of digital circuits/systems, cryptoeconomics, decentralized blockchain technology, and combinatorics. He has received several teaching awards, a best paper award and a Ten Year Best Paper Retrospective Award. He is the author of one book and he holds several patents. He is a member of IEEE, ACM, AMS, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society

Aron Laszka is an Assistant Professor in the College of Information Sciences and Technology at the Pennsylvania State University and a Scientist at NTT Research. His research interests revolve around artificial intelligence, including machine learning; cyber-physical system, with a focus on societal-scale systems; and cybersecurity. Previously, he was an Assistant Professor at the University of Houston from 2017 to 2022, a Research Assistant Professor at Vanderbilt University from 2016 to 2017, and a Postdoctoral Scholar at the University of California, Berkeley from 2015 to 2016. He graduated summa cum laude with a Ph.D. in Computer Science from the Budapest University of Technology and Economics in 2014.