Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts

Pedro Antonino[1], Juliandson Ferreira[2], Augusto Sampaio[2], and A. W. Roscoe[1,3,4]

[1] The Blockhouse Technology Limited, Oxford, UK
pedro@tbtl.com
[2] Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil
jef@cin.ufpe.br, acas@cin.ufpe.br
[3] Department of Computer Science, Oxford University, Oxford, UK
[4] University College Oxford Blockchain Research Centre, Oxford, UK
awroscoe@gmail.com

Abstract. Smart contracts are the building blocks of the "code is law" paradigm: the smart contract's code indisputably describes how its assets are to be managed - once it is created, its code is typically immutable. Faulty smart contracts present the most significant evidence against the practicality of this paradigm; they are well-documented and resulted in assets worth vast sums of money being compromised. To address this issue, the Ethereum community proposed (i) tools and processes to audit/analyse smart contracts, and (ii) design patterns implementing a mechanism to make contract code mutable. Individually, (i) and (ii) only partially address the challenges raised by the "code is law" paradigm. In this paper, we combine elements from (i) and (ii) to create a systematic framework that moves away from "code is law" and gives rise to a new "specification is law" paradigm. It allows contracts to be created and upgraded but only if they meet a corresponding formal specification. The framework is centered around *a trusted deployer*: an off-chain service that formally verifies and enforces this notion of conformance. We have prototyped this framework, and investigated its applicability to contracts implementing two widely used Ethereum standards: the ERC20 Token Standard and ERC1155 Multi Token Standard, with promising results.

Keywords: Formal Verification · Smart Contracts · Ethereum · Solidity · Safe Deployment · Safe Upgrade

1 Introduction

A *smart contract* is a stateful reactive program that is stored in and processed by a trusted platform, typically a blockchain, which securely executes such a program and safely stores its persistent state. Smart contracts were created to provide an unambiguous, automated, and secure way to manage digital assets. They are the building blocks of the "code is law" paradigm, indisputably describing how their assets are to be managed. To implement this paradigm, many

smart contract platforms - including Ethereum, the platform we focus on - disallow the code of a contract to be changed once deployed, effectively enforcing a notion of *code/implementation immutability*.

Implementation immutability, however, has two main drawbacks. Firstly, contracts cannot be patched if the implementation is found to be incorrect after being deployed. There are many examples of real-world contract instances with flaws that have been exploited with astonishing sums of cryptocurrencies being taken over [37,44,8]. The ever-increasing valuation of these assets presents a significant long-standing incentive to perpetrators of such attacks. Secondly, contracts cannot be optimised. The execution of a contract function has an explicit cost to be paid by the caller that is calculated based on the contract's implementation. Platform participants would, then, benefit from contracts being updated to a functionally-equivalent but more cost-effective implementation, which is disallowed by this sort of code immutability.

To overcome this limitation, the Ethereum community has adopted the *proxy pattern* [38] as a mechanism by which one can mimic contract upgrades. The simple application of this pattern, however, presents a number of potential issues. Firstly, the use of this mechanism allows for the patching of smart contracts but it does not address the fundamental underlying problem of correctness. Once an issue is detected, it can be patched but (i) it may be too late, and (ii) what if the patch is faulty too? Secondly, it typically gives an, arguably, unreasonable amount of power to the maintainers of this contract. These special contract users can change the contract's code with little oversight. The main flaw of such an approach is, arguably, the fact that no guarantees are enforced by this updating process; the contract implementations can change rather arbitrarily as long as the right participants have approved the change. In such a context, the "code is law" paradigm is in fact nonexistent.

To address these issues, we propose a *systematic deployment framework* that requires contracts to be formally verified before they are created and upgraded; we target the Ethereum platform and smart contracts written in Solidity. We propose a *verification framework* based on the *design-by-contract methodology* [28]. The specification format that we propose is similar to what the community has used, albeit in an informal way, to specify the behaviour of common Ethereum contracts [43]. Our framework also relies on our own version of the proxy pattern to carry out updates but in a sophisticated and safe way. We rely on a *trusted deployer*, which is an off-chain service, to vet contract creations and updates. These operations are only allowed if the given implementation meets the expected specification - the contract specification is set at the time of contract creation and remains unchanged during its lifetime. As an off-chain service, our framework can be readily and efficiently integrated into existing blockchain platforms, but the same is not true of an on-chain implementation; we elaborate on this trade-off in Section 3. Participants can also check whether a contract has been deployed via our framework so that they can be certain the contract they want to execute has the expected behaviour.

Our framework promotes a paradigm shift where the specification is immutable instead of the implementation/code. Thus, it moves away from "code is law" and proposes the *"specification is law" paradigm* - enforced by formal verification. This new paradigm addresses all the concerns that we have highlighted: arbitrary code updates are forbidden as only conforming implementations are allowed, and buggy contracts are prevented from being deployed as they are vetted by a formal verifier. Thus, contracts can be optimised and changed to meet evolving business needs and yet contract stakeholders can rely on the guarantee that the implementations always conform to their corresponding specifications. As specifications are more stable and a necessary element for assessing the correctness of a contract, we believe that a framework that focuses on this key artifact and makes it immutable improves on the current "code is law" paradigm. We further discuss the benefits of this paradigm shift in Section 6.

We have created a prototype of our framework, and conducted a case study that investigates its applicability to real-world smart contracts implementing the widely used ERC20 and ERC1155 Ethereum token standards [43,34]. We analysed specifically how the sort of formal verification that we use fares in handling practical contracts and obtained promising results.

In this paper, we assume the deployer is a trusted third party and focus on the functional aspect of our framework. We are currently working on an implementation of the trusted deployer that relies on a Trusted Execution Environment (TEE) [27], specifically the AMD SEV implementation [36].

Outline. Section 2 introduces the relevant background material. Section 3 introduces our framework, and Section 4 the evaluation that we conducted. Section 5 discusses related work, whereas Section 6 presents our concluding remarks.

2 Background

2.1 Ethereum and Solidity

The Ethereum blockchain is arguably the most popular smart contract platform. A *participant* in Ethereum controls *addresses* in the blockchain, each of which has a balance of Ether - Ethereum's cryptocurrency - associated with it. These addresses are akin to account numbers in traditional banking. A participant that controls an address also controls the balance associated to it. Thus, they can send a *transaction* to Ethereum requesting the transfer of some amount of the balance associated to one of its addresses. Aside from these addresses that are managed by external entities, Ethereum also allows addresses to be managed by a *program* (a smart contract). In addition to a balance, these *smart contract* addresses have some code and data associated to them. While the former defines the functions offered by the contract, the latter captures its persistent state. For a detailed presentation of Ethereum, see, for instance, [1,2].

Solidity is arguably the most used language for writing smart contracts. A contract in Solidity is a concept very similar to that of a *class* in object-oriented languages, and a contract instance a sort of long-lived persistent object. We introduce the main elements of Solidity using the `ToyWallet` contract in Figure 1.

It implements a very basic "wallet" contract that participants and other contracts can rely upon to store their Ether. The *member variables* of a contract define the persistent state of the contract. This example contract has a single member variable `accs`, a mapping from addresses to 256-bit unsigned integers, which keeps track of the balance of Ether each "client" of the contract has in the `ToyWallet`; the integer `accs[addr]` gives the current balance for address `addr`, and an address is represented by a 160-bit number.

Public functions describe the operations that participants and other contracts can execute on the contract. The contract in Figure 1 has *public functions* `deposit` and `withdraw` that can be used to transfer Ether into and out of the `ToyWallet` contract, respectively. In Solidity, functions have the implicit argument `msg.sender` designating the caller's address, and `payable` functions have the `msg.value` which depict how much *Wei* - the most basic (sub)unit of Ether - is being transferred, from caller to callee, with that function invocation; such a transfer is carried out implicitly by Ethereum. For instance, when `deposit` is called on an instance of `ToyWallet`, the caller can decide on some amount `amt` of Wei to be sent with the invocation. By the time the `deposit` body is about to execute, Ethereum will already have carried out the transfer from the balance associated to the caller's address to that of the `ToyWallet` instance - and `amt` can be accessed via `msg.value`. Note that, as mentioned, this balance is part of the blockchain's state rather than an explicit variable declared by the contract's code. One can programmatically access this implicit balance variable for address `addr` with the command `addr.balance`. Solidity's construct `require (condition)` aborts and reverts the execution of the function in question if `condition` does not hold - even in the case of implicit Ether transfers. The call `addr.send(amount)` sends `amount` Wei from the currently executing instance to address `addr`; it returns `true` if the transfer was successful, and `false` otherwise. For instance, the first `require` statement in the function `withdraw` requires the caller to have the funds they want to withdraw, whereas the second requires the `msg.sender.send(value)` statement to succeed, i.e. the `value` must have been correctly withdrawn from `ToyWallet` to `msg.sender`. The final statement in this function updates the account balance of the caller (i.e. `msg.sender`) in `ToyWallet` to reflect the withdrawal.

We use the transaction *create-contract* as a means to create an instance of a Solidity smart contract in Ethereum. In reality, Ethereum only accepts contracts in the *EVM bytecode* low-level language - Solidity contracts need to be compiled into that. The processing of a transaction *create-contract*$(c, args)$ creates an instance of contract c and executes its constructor with arguments $args$. Solidity contracts without a constructor (as our example in Figure 1) are given an implicit one. A *create-contract* call returns the address at which the contract instance was created. We omit the *args* when they are not relevant for a call. We use σ to denote the state of the blockchain where $\sigma[ad].balance$ gives the balance for address ad, and $\sigma[ad].storage.mem$ the value for member variable mem of the contract instance deployed at ad for this state. For instance, let $c_{\texttt{tw}}$ be the code in Figure 1, and $\texttt{addr}_{\texttt{tw}}$ the address returned by the processing of

```
contract ToyWallet {
  mapping (address => uint) accs;

  function deposit () payable public {
    accs[msg.sender] = accs[msg.sender] + msg.value;
  }

  function withdraw (uint value) public {
    require(accs[msg.sender] >= value);
    bool ok = msg.sender.send(value);
    require(ok);
    accs[msg.sender] = accs[msg.sender] - value;
  }
}
```

Fig. 1: ToyWallet contract example.

create-contract($c_{\texttt{tw}}$). For the blockchain state σ' immediately after this processing, we have that: for any address *addr*, $\sigma'[\texttt{addr}_{\texttt{tw}}].storage.accs[addr] = 0$ and its balance is zero, i.e., $\sigma'[addr_{\texttt{tw}}].balance = 0$. We introduce and use this intuitive notation to present and discuss state changes as it can concisely and clearly capture them. There are many works that formalise such concepts [18,45,7].

A transaction *call-contract* can be used to invoke contract functions; processing *call-contract*$(addr, func\_sig, args)$ executes the function with signature *func_sig* at address *addr* with input arguments *args*. When a contract is created, the code associated with its non-constructor public functions is made available to be called by such transactions. The constructor function is only run (and available) at creation time. For instance, let $\texttt{addr}_{\texttt{tw}}$ be a fresh `ToyWallet` instance and `ToyWallet.deposit` give the signature of the corresponding function in Figure 1, processing the transaction *call-contract*($\texttt{addr}_{\texttt{tw}}$, `ToyWallet.deposit`, *args*) where $args = \{msg.sender = \texttt{addr}_{snd}, msg.value = 10\}$ would cause the state of this instance to be updated to σ'' where we have that $\sigma''[\texttt{addr}_{\texttt{tw}}].storage.accs[\texttt{addr}_{snd}] = 10$ and $\sigma''[\texttt{addr}_{\texttt{tw}}].balance = 10$. So, the above transaction has been issued by address $\texttt{addr}_{snd}$ which has transferred 10 Wei to $\texttt{addr}_{tw}$.

2.2 Formal verification with *solc-verify*

The modular verifier *solc-verify* [17,16] was created to help developers to formally check that their Solidity smart contracts behave as expected. Input contracts are manually annotated with contract *invariants* and their functions with *pre-* and *postconditions*. An annotated Solidity contract is then translated into a Boogie program which is verified by the Boogie verifier [9,21]. Its modular nature means that *solc-verify* verifies functions locally/independently, and function calls are abstracted by the corresponding function's specification, rather than their implementation being precisely analysed/executed. These specification constructs have their typical meaning. An invariant is valid if it is established by the constructor and maintained by the contract's public functions, and a function meets its specification if and only if from a state satisfying its pre-conditions, any state

```
/**
* @notice postcondition address(this).balance == __verifier_old_uint(address(
    this).balance) - value
* @notice postcondition accs[msg.sender] == __verifier_old_uint(accs[msg.
    sender]) - value
* @notice postcondition forall (address addr) addr == msg.sender ||
    __verifier_old_uint(accs[addr]) == accs[addr]
*/
function withdraw (uint value) public {
  require(accs[msg.sender] >= value);
  bool ok;
  (ok,) = msg.sender.call.value(value)("");
  require(ok);
  accs[msg.sender] = accs[msg.sender] - value;
}
```

Fig. 2: `ToyWallet` alternate buggy `withdraw` implementation with specification.

successfully terminating respects its postconditions. So the notion is that of partial correctness. Note that an aborted and reverted execution, such as one triggered by a failing `require` command, does not successfully terminate. We use Figure 2 illustrates a *solc-verify* specification for an alternative version of the `ToyWallet`'s `withdraw` function. The postconditions specify that the balance of the instance and the wallet balance associated with the caller must decrease by the withdrawn amount and no other wallet balance must be affected by the call.

This alternative implementation uses `msg.sender.call.value(value)("")` instead of `msg.sender.send(value)`. While the latter only allows for the transfer of `value` Wei from the instance to address `msg.sender`, the former *delegates control* to `msg.sender` in addition to the transfer of `value`.[5] If `msg.sender` is a smart contract instance that calls `withdraw` again during this control delegation, it can withdraw all the funds in this alternative `ToyWallet` instance - even the funds that were not deposited by it. This *reentrancy* bug is detected by *solc-verify* when it analyses this alternative version of the contract. A similar bug was exploited in what is known as the DAO attack/hack to take over US$53 million worth of Ether [37,44,8].

3 Safe Ethereum Smart Contracts Deployment

We propose a framework for the *safe creation and upgrade of smart contracts* based around a *trusted deployer*. This entity is trusted to only create or update contracts that have been verified to meet their corresponding specifications. A smart contract development process built around it prevents developers from deploying contracts that have not been implemented as intended. Thus, stakeholders can be sure that contract instances deployed by this entity, even if their code is upgraded, comply with the intended specification.

[5] In fact, the function `send` also delegates control to `msg.sender` but it does in such a restricted way that it cannot perform any relevant computation. So, for the purpose of this paper and to simplify our exposition, we ignore this delegation.

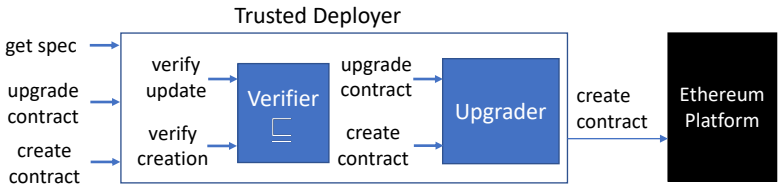

Fig. 3: Trusted deployer architecture.

Our trusted deployer targets the Ethereum platform, and we implement it as an off-chain service. Generally speaking, a trusted deployer could be implemented as a smart contract in a blockchain platform, as part of its consensus rules, or as an off-chain service. In Ethereum, implementing it as a smart contract is not practically feasible as a verification infrastructure on top of the EVM [1] would need to be created. Furthermore, blocks have an upper limit on the computing power they can use to process their transactions, and even relatively simple computing tasks can exceed this upper limit [46]. As verification is a notoriously complex computing task, it should exceed this upper limit even for reasonably small systems. Neither can we change the consensus rules for Ethereum.

We present the architecture of the *trusted deployer infrastructure* in Figure 3. The trusted deployer relies on an internal *verifier* that implements the functions *verify-creation*$_{\sqsubseteq}$ and *verify-upgrade*$_{\sqsubseteq}$, and an *upgrader* that implements functions *create-contract* and *upgrade-contract*; we detail what these functions do in the following. The deployer's *create-contract* (*upgrate-contract*) check that an implementation meets its specification by calling *verify-creation*$_{\sqsubseteq}$ (*verify-upgrade*$_{\sqsubseteq}$) before relaying this call to the upgrader's *create-contract* (*upgrade-contract*) which effectively creates (upgrades) the contract in the Ethereum platform. The *get-spec* function can be used to test whether a contract instance has been deployed by the trusted deployer and which specification it satisfies.

The *verifier* is used to establish whether an implementation meets a specification. A verification framework is given by a triple $(\mathcal{S}, \mathcal{C}, \sqsubseteq)$ where $\mathcal{S}$ is a language of smart contract specifications, $\mathcal{C}$ is a language of implementations, and $\sqsubseteq \in (\mathcal{S} \times \mathcal{C})$ is a satisfiability relation between smart contracts' specifications and implementations. In this paper, $\mathcal{C}$ is the set of Solidity contracts and $\mathcal{S}$ a particular form of Solidity contracts, possibly annotated with contract invariants, that include function signatures annotated with postconditions. The functions *verify-creation*$_{\sqsubseteq}$ and *verify-upgrade*$_{\sqsubseteq}$ both take a specification $s \in \mathcal{S}$ and a contract implementation $c \in \mathcal{C}$ and test whether c meets s - they work in slightly different ways as we explain later. When an implementation does not meet a specification, verifiers typically return an error report that points out which parts of the specification do not hold and maybe even witnesses/-counterexamples describing system behaviours illustrating such violations; they provide valuable information to help developers correct their implementations.

The upgrader is used to create and manage *upgradable smart contracts* - Ethereum does not have built-in support for contract upgrades. Function *create-contract* creates an upgradable instance of contract $c \in \mathcal{C}$ - it returns the Ethereum address where the instance was created - whereas *upgrade-contract* allows for the contract's behaviour to be upgraded. The specification used for a successful contract creation will be stored and used as the intended specification for future upgrades. Only the creator of a *trusted contract* can update its implementation.

Note that once a contract is created via our trusted deployer, the instance's specification is fixed, and not only its initial implementation but all upgrades are guaranteed to satisfy this specification. Therefore, participants in the ecosystem interacting with this contract instance can be certain that its behaviour is as intended by its developer during the instance's entire lifetime, even if the implementation is upgraded as the contract evolves.

In this paper, we focus on contract upgrades that preserve the signature of public functions. Also, we assume contract specifications fix the data structures used in the contract implementation. However, we plan to relax these restrictions in future versions of the framework, allowing the data structures in the contract implementation to be a data refinement of those used in the specification; we also plan to allow the signature of the implementation to extend that of the specification, provided some notion of behaviour preservation is obeyed when the extended interface is projected into the original one.

3.1 Verifier

We propose *design-by-contract* [28] as a methodology to specify the behaviour of smart contracts. In this traditional specification paradigm, conceived for object-oriented languages, a developer can specify invariants for a class and pre-/postconditions for its methods. Invariants must be established by the constructor and guaranteed by the public methods, whereas postconditions are ensured by the code in the method's body provided that the pre-conditions are guaranteed by the caller code and the method terminates - currently, we focus on partial correctness. We propose a specification format that defines what the member variables and signatures of member functions should be. Additionally, the function signatures can be annotated with postconditions, and the specification with invariants; these annotations capture the expected behaviour of the contract. In ordinary programs, a function is called in specific *call sites* fixed in the program's code. Pre-conditions can, then, be enforced and checked in these call sites. In the context of public functions of smart contracts, however, any well-formed transaction can be issued to invoke such a function. Hence, we move away from preconditions in our specification, requiring, thus, postconditions to be met whenever public functions successfully terminate.

Figure 4 illustrates a specification for the `ToyWallet` contract. Invariants are declared in a comment block preceding the contract declaration, and function postconditions are declared in comment blocks preceding their signatures. Our specification language reuses constructs from Solidity and the *solc-verify*

```
/**
* @notice invariant accs[address(this)] == 0
*/
contract ToyWallet {
  mapping (address => uint) accs;

  /**
  * @notice postcondition forall (address addr) accs[addr] == 0
  */
  constructor() public;

  /**
  * @notice postcondition address(this).balance == __verifier_old_uint(
      address(this).balance) + msg.value
  * @notice postcondition accs[msg.sender] == __verifier_old_uint(accs[msg.
      sender]) + msg.value
  * @notice postcondition forall (address addr) addr == msg.sender ||
      __verifier_old_uint(accs[addr]) == accs[addr]
  */
  function deposit () payable public;

  /**
  * @notice postcondition address(this).balance == __verifier_old_uint(
      address(this).balance) - value
  * @notice postcondition accs[msg.sender] == __verifier_old_uint(accs[msg.
      sender]) - value
  * @notice postcondition forall (address addr) addr == msg.sender ||
      __verifier_old_uint(accs[addr]) == accs[addr]
  */
  function withdraw (uint value) public;
}
```

Fig. 4: ToyWallet specification.

specification language, which in turn borrows elements from the Boogie language [9,21]. Member variables and function signature declarations are as prescribed by Solidity, whereas the conditions on invariants, and postconditions are side-effect-free Solidity expressions extended with quantifiers and the expression `__verifier_old_`x`(v)` that can only be used in a postcondition, and it denotes the value of `v` in the function's execution pre-state.

We choose to use Solidity as opposed to EVM bytecode as it gives a cleaner semantic basis for the analysis of smart contracts [7] and it also provides a high-level error message when the specification is not met. The satisfiability relation $\sqsubseteq$ that we propose is as follows.

Definition 1. *The relation $s \sqsubseteq c$ holds iff:*

- Syntactic obligation*: a member variable is declared in s if and only if it is declared in c with the same type, and they must be declared in the same order. A public function signature is declared in s if and only if it is declared and implemented in c.*
- Semantic obligation*: invariants declared in s must be respected by c, and the implementation of functions in c must respect their corresponding guards and postconditions described in s.*

The purpose of this paper is not to provide a formal semantics to Solidity or to formalise the execution model implemented by the Ethereum platform. Other works propose formalisations for Solidity and Ethereum [17,6,45]. Our focus is on using the modular verifier *solc-verify* to discharge the semantic obligations imposed by our satisfaction definition.

The *verify-creation*$_{\sqsubseteq}$ function works as follows. Firstly, the syntactic obligation imposed by Definition 1 is checked by a syntactic comparison between s and c. If it holds, we rely on *solc-verify* to check whether the semantic obligation is fulfilled. We use what we call a *merged contract* as the input to *solc-verify* - it is obtained by annotating c with the corresponding invariants and postconditions in s. If *solc-verify* is able to discharge all the proof obligations associated to this merged contract, the semantic obligations are considered fulfilled, and *verify-creation*$_{\sqsubseteq}$ succeeds.

Function *verify-upgrade*$_{\sqsubseteq}$ is implemented in a very similar way but it relies on a slightly different satisfiability relation and merged contract. While *verify-creation*$_{\sqsubseteq}$ checks that the obligations of the constructor are met by its implementation, *verify-upgrade*$_{\sqsubseteq}$ *assumes* they do, since the constructor is only executed - and, therefore, its implementation checked for satisfiability - at creation time. The upgrade process only checks conformance for the implementation of the (non-constructor) public functions.

3.2 Upgrader

Ethereum does not provide a built-in mechanism for upgrading smart contracts. However, one can simulate this functionality using the *proxy pattern* [38], which splits the contract across two instances: the *proxy instance* holds the persistent state and the upgrade logic, and rely on the code in an *implementation instance* for its business logic. The proxy instance is the *de-facto* instance that is the target of calls willing to execute the upgradable contract. It stores the address of the implementation instance it relies upon, and the behaviour of the proxy's public functions can be upgraded by changing this address. Our *upgrader* relies on our own version of this pattern to deploy *upgradable* contracts.

Given a contract c that meets its specification according to Definition 1, the upgrader creates the Solidity contract $proxy(c)$ as follows. It has the same member variable declarations, in the same order, as c - having the same order is an implementation detail that is necessary to implement the sort of delegation we use as it enforces proxy and implementation instances to share the same memory layout. In addition to those, it has a new *address* member variable called `implementation` - it stores the address of the implementation instance. The constructor of $proxy(c)$ extends the constructor of c with an initial setting up of the variable `implementation`.[6] This proxy contract also has a public function

[6] Instead of using the proxy pattern `initialize` function to initialise the state of the proxy instance, we place the code that carries out the desired initialisation directly into the proxy's constructor. Our approach benefits from the inherent behaviour of constructors - which only execute once and at creation time - instead of having to

`upgrade` that can be used to change the address of the implementation instance. The trusted deployer is identified by a trusted Ethereum address $\texttt{addr}_{td}$. This address is used to ensure calls to `upgrade` can *only* be issued by the trusted deployer. In the process of creating and upgrading contracts the trusted deployer acts as an external participant of the Ethereum platform. We assume that the contract implementations and specifications do not have member variables named `implementation`, or functions named `upgrade` to avoid name clashes.

The proxy instance relies on the low-level `delegatecall` Solidity command to dynamically borrow and execute the function implementations defined in the contract instance at `implementation`. When the contract instance at address `proxy` executes `implementation.delegatecall(`sig`, args)`, it executes the code associated with the function with signature *sig* stored in the instance at address `implementation` but applied to the `proxy` instance - modifying its state - instead of `implementation`. For each (non-constructor) public function in c with signature *sig*, $proxy(c)$ has a corresponding function declaration whose implementation relies on `implementation.delegatecall(`sig`, args)`. This command was proposed as a means to implement and deploy contracts that act as a sort of dynamic library. Such a contract is deployed with the sole purpose of other contracts borrowing and using their code.

The upgrader function *create-contract*(c) behaves as follows. Firstly, it issues transaction *create-contract*($c, args$) to the Ethereum platform to create the initial implementation instance at address $\texttt{addr}_{impl}$. Secondly, it issues transaction *create-contract*($proxy(c), args$), such that `implementation` would be set to $\texttt{addr}_{impl}$, to create the proxy instance at address $\texttt{addr}_{px}$. Note that both of these transactions are issued by and using the trusted deployer's address $\texttt{addr}_{td}$. The upgrader function *upgrade-contract*(c) behaves similarly, but the second step issues transaction *call-contract*($\texttt{addr}_{px}$,`upgrade`, $args$), triggering the execution of function `upgrade` in the proxy instance and changing its `implementation` address to the new implementation instance.

4 Evaluation

4.1 Context

The presentation of our evaluation is structured into three main sections: first we describe the context (Section 4.1), then we provide the results (Section 4.2) and, finally, we discuss the threats to vaidity and the limitations (Section 4.3). The main purpose of the study is to provide evidence that the proposed framework can be used to ensure the deployment and evolution of smart contracts in a secure manner. The framework must be integrated into a development process and whenever there is any change in the code, one must verify if the change meets the specification, so we intend to answer the following research question:

implement this behaviour for the non-constructor function `initialize`. Our Trusted Deployer, available at `https://github.com/stanis18/safeeveolution`, automatically generates the code for such a proxy.

Does the proposed framework improve the process of creation and evolving smart contracts?

In the first phase, a review of the literature was carried out. The objective was to explore the main features of smart contract development patterns and the most common error types. As result, we could identify opportunities for the application of the framework in line with the objectives of the study. In the second phase, we identified, documented and validated requirements related to two Ethereum smart contract specifications: ERC20 and ERC1155. We chose these standards because they are widely used and in an advanced state of maturity. By structuring the existing requirements in natural language we were able to extract the formal properties used in the verification process and create a corresponding formal contract specification. Then we conducted a quantitative analysis, in order to verify the feasibility of the framework, to evaluate how it could be integrated into a development process and check the effectiveness, which can be measured by the number of errors found and efficiency measured by the time to process the verification.

We have implemented a tool to check the syntactic and semantic obligations imposed by our framework. It relies on the abstract syntax tree generated by the Solidity compiler [3] to check the syntactic obligations, and to create a contract resulting from merging the solidity smart contract with the corresponding formal specification; this is then checked by the *solc-verify* as a means to discharge the semantic obligations. We applied our framework to a number of real-world Solidity smart contract samples implementing the ERC20 and the ERC1155 token standards to evaluate the benefits it brings. Each of these standards defines a contract interface and is accompanied by an informal description of its functions' behaviours. The contract samples we analysed were extracted from twelve github repositories that were public, and presented reasonably complex commits that changed the smart contract behavior. The samples also cover aspects of evolution that are related to improving the readability and maintenance of the code, but also optimisations where, for instance, redundant checks executed by a function were removed. The evaluation was carried out on a Lenovo IdeapadGaming3i with the operational system Windows 10, Intel(R) Core(TM) i7-10750 CPU @ 2.60GHz, 8GB of RAM, with Docker Engine 20.15.5 and Solidity compiler version 0.5.17.

4.2 Results

Our framework was able to identify errors of the following categories: Integer Overflow and Underflow (IOU); Nonstandard Token Interface (NTI), when the contract does not meet the syntactic restriction defined by the standard; wrong operator (WOP), for instance, when the $<$ operator would be expected but $\leq$ is used instead; and Verification Error (VRE), when the verification process cannot be completed or the results were inconclusive. It also established conformance for some of the samples analysed. Table 1 shows the the complete list of all results we obtained. In the following, we use some snippets extracted from ERC20 examples

ERC20							
Repository	Commit	Time	Output	Repository	Commit	Time	Output
0xMonorepo	548fda	2.85s	WOP	DsToken	3c436c	3.77s	No errors
0xMonorepo	6f2cb6	2.84s	No errors	DsToken	733e5c	3.81s	No errors
0xMonorepo	c84be8	2.57s	No errors	DsToken	8b8263	3.08s	No errors
Ambrosus	9fb24b	3.15s	No errors	Klenergy	3d4d62	5.14s	No errors
Ambrosus	b1806b	2.99s	No errors	Klenergy	60263d	1.70s	VRE
Ambrosus	db3ea0	3.74s	No errors	OpenZeppelin	3a5da7	3.59s	No errors
DigixDao	0550e8	5.97s	No errors	OpenZeppelin	43ebb4	3.57s	No errors
DigixDao	1c0c4f	8.82s	No errors	OpenZeppelin	5db741	3.87s	No errors
DigixDao	5aee64	7.60s	NTI	OpenZeppelin	5dfe72	3.96s	No errors
DigixDao	6bddc6	7.74s	No errors	OpenZeppelin	9b3710	3.45s	No errors
DigixDao	845F03	9.17s	No errors	Uniswap	4e4546	3.67s	No errors
DigixDao	aabf24	2.97s	No errors	Uniswap	55ae25	3.43s	WOP
DigixDao	e221ff	9.21s	No errors	Uniswap	e382d7	3.57s	IOU
DigixDao	e320a2	8.89s	No errors	SkinCoin	25db99	0.99s	NTI
DsToken	08412f	4.14s	WOP	SkinCoin	27c298	1.94s	NTI
DsToken	10c964	3.66s	No errors	SkinCoin	ac33d8	3.23s	No errors

ERC1155							
0xSequence	319740	4.82s	No errors	OpenZeppelin	0db76e	5.59s	No errors
0xSequence	578d46	5.31s	No errors	OpenZeppelin	440b65	6.61s	No errors
0xSequence	99012f	5.59s	No errors	OpenZeppelin	5db741	6.70s	No errors
0xSequence	acfa7c	5.81s	No errors	OpenZeppelin	956d66	8.58s	No errors
Desc-Stock	44464c	4.34s	IOU	Ejin-Erc	30dba0	4.13s	No errors
Desc-Stock	4c5d80	5.18s	IOU	Ejin-Erc	614714	4.49s	No errors
Desc-Stock	96d5b2	4.33s	WOP	Ejin-Erc	bf4d04	4.01s	No errors
Desc-Stock	ae8a13	4.24s	IOU	Ejin-Erc	cc96af	4.57s	No errors
Desc-Stock	bf2c1a	3.60s	IOU	Ejin-Erc	e20fc9	3.77s	No errors

Table 1: ERC20 and ERC1155 Results

in our evaluation to illustrate the application of our framework, some of the errors that we found, and a case of a safe evolution.

The ERC20 is likely to be the most widely implemented Ethereum standard. It defines member variables: `totalSupply` keeps track of the total number of tokens in circulation, `balanceOf` maps a wallet (i.e. address) to the balance it owns, and `allowance` stores the number of tokens that an address has made available to be spent by another one. It defines public functions: `totalSupply`, `balanceOf` and `allowance` are accessors for the above variables; `transfer` and `transferFrom` can be used to transfer tokens between contracts; and `approve` allows a contract to set an "allowance" for a given address.

Figure 5 presents a reduced specification, focusing on functions `transferFrom` and `allowance` for the purpose of this discussion, derived from the informal description in the standard [43]. In Line 1, we define an invariant requiring that the total number of tokens remain unchanged regardless of the operation carried out by the contract. The function `allowance` does not change the state of the

```
1  /// @notice invariant totalSupply == __verifier_sum_uint(balanceOf)
2  contract IERC20  {
3  uint256 totalSupply;
4  mapping (address => uint256) balanceOf;
5  mapping (address => mapping (address => uint256)) allowance;
6
7  //... functions transfer, totalSupply, balanceOf, and approve omitted ...
8
9  /// @notice postcondition allowance[owner][spender] == remaining
10 function allowance(address owner, address spender) external view returns (
       uint256 remaining);
11
12 /**
13 * @notice postcondition ( ( balanceOf[_from] == __verifier_old_uint (
       balanceOf[_from] ) - _value && _from != _to )  ||  ( balanceOf[_from] ==
       __verifier_old_uint ( balanceOf[_from] ) && _from == _to ) && success )
       || !success
14 * @notice postcondition ( ( balanceOf[_to] == __verifier_old_uint ( balanceOf
       [_to] ) + _value && _from != _to ) ||  ( balanceOf[_to] ==
       __verifier_old_uint ( balanceOf[_to] ) && _from == _to ) && success )  ||
        !success
15 * @notice postcondition allowance[_from][msg.sender] == __verifier_old_uint (
        allowance[_from][msg.sender] ) - _value || _from == msg.sender
16 * @notice postcondition allowance[_from][msg.sender] <= __verifier_old_uint (
        allowance[_from][msg.sender] ) || _from == msg.sender
17 */
18 function transferFrom(address _from, address _to, uint256 _value) public
       returns (bool success);
19 }
```

Fig. 5: ERC20 reduced specification

smart contract so it has only one postcondition (line 9) to ensure that it will return the correct amount of tokens available for withdrawals. The `transferFrom` function has 4 postconditions; the operation is successful only when the tokens are debited from the source account and credited in the destination account, according to the specifications provided in the ERC20 standard. The first two postconditions (lines 13 to 14) require that the balances are updated as expected, whereas the purpose of the last two (lines 15 to 16) is to ensure that the tokens available for withdrawal have been properly updated.

We use the snippet in Figure 6 - extracted from the Uniswap repository, commit 55ae25 - to illustrate the detection of wrong operator errors. When checked by our framework, the third postcondition for the `transferFrom` function presented in the specification in Figure 5 is not satisfied. Note that the allowance amount is not debited if the amount to be transferred is equal to the maximum integer supported by Solidity (i.e. `uint(-1)`). A possible solution would consist of removing the `if` branching, allowing the branch code to always execute.

The code snippet in Figure 7 - DigixDao repository, commit 5aee64 - does not conform to its formal specification. The correct allowance for the spender is only returned when it is not greater than the owner's balance. To fix this issue, we need to remove all code related to `_balance`, ensuring that the `_allowance` will be returned regardless of the `_balance` amount.

```
function transferFrom(address from, address to, uint value) external
returns (bool success) {
    if (allowance_[from][msg.sender] != uint(-1)) {
        allowance_[from][msg.sender] =
        allowance_[from][msg.sender].sub(value);
    }
    _transfer(from, to, value);
    return true;
}
```

Fig. 6: Buggy ERC20 transferFrom function

The ERC1155 was created in order to promote a better integration between the ERC20 and ERC721 standards. It provides an interface for managing any combination of fungible and non-fungible tokens in a single contract efficiently. The functions `balanceOf` and `balanceOfBatch` returns the balance of specific tokens of the address or a list of addresses specified in the parameter function respectively. The `isApprovedForAll` return a boolean value informing if an address is allowed to handle the tokens from another address. The `safeTransferFrom` method transfers tokens in a safe way to a valid ERC1155 address. The `transferFrom` function can do batch operations, transfering tokens to several wallets at the same time, reducing transaction costs and minimizing impacts on the network. The `setApprovalForAll` method gives an address permission to handle another address' tokens.

The snippet in Figure 10 - extracted from Descentralized-Stock repository, commit 96d5b2 - o illustrates another example of the wrong operator error. A postcondition was not satisfied, because according to the specification the size of the `_ids` and `_values` arrays must be equal. So any call to this function would result in a error or unexpected behavior. A possible solution would consist of changing the operator `!=` for `==`.

Figures 8 and 9 - extracted from the 0xSequence-erc-1155 repository, commits 99012f and 319740, respectively - illustrate a case of safe contract evolution. The code of this contract has undergone significant changes. The refactoring in question is one of the most common and is known as extract method

```
function allowance(address _owner, address _spender) public returns (uint256
      remaining) {
    uint256 _allowance = allowed[_owner][_spender];
    uint256 _balance = balances[_owner];
    if (_allowance > _balance) {
       remaining = _balance;
    } else {
       remaining =  _allowance;
    }
    return remaining;
}
```

Fig. 7: Buggy ERC20 allowance function

```
function safeTransferFrom(address _from, address _to, uint256 _id, uint256
    _value, bytes memory _data) public {

    require((msg.sender == _from) || operators[_from][msg.sender], "
        INVALID_OPERATOR");
    require(_to != address(0),"INVALID_RECIPIENT");
    require(_value >= balances[_from][_id]) is not necessary since checked
        with safemath operations

    _safeTransferFrom(_from, _to, _id, _value, _data);
  }
```

Fig. 8: safeTransferFrom function before refactoring

(function, in Solidity). From commit 319740 to 99012f, the new internal function `_callonERC1155Received` was created, and the extracted code from the `_safeTransferFrom` was moved into it.

The results of our evaluation suggest that the kind of verification that we employ in our framework is tractable, as *solc-verify* can efficiently analyse these

```
function safeTransferFrom(address _from, address _to, uint256 _id, uint256
    _amount, bytes memory _data) public {

    require((msg.sender == _from) || operators[_from][msg.sender], "ERC1155#
        safeTransferFrom: INVALID_OPERATOR");
    require(_to != address(0),"ERC1155#safeTransferFrom: INVALID_RECIPIENT");
    require(_amount >= balances[_from][_id]) is not necessary since checked
        with safemath operations

    _safeTransferFrom(_from, _to, _id, _amount);
    _callonERC1155Received(_from, _to, _id, _amount, _data);
  }
```

Fig. 9: Successful refactoring of the safeTransferFrom function

```
  function safeBatchTransferFrom(address _from, address _to, uint256[]
      calldata _ids, uint256[] calldata _values, bytes calldata _data)
      external {
        require(_to != address(0) && _from != address(0));
        require(_ids.length != _values.length);
        require(_approv[_from][msg.sender] || _from == msg.sender);

        for (uint256 i = 0; i < _ids.length; ++i) {
            require(_balance[_from][_ids[i]] >= _values[i]);
            _balance[_from][_ids[i]] -= _values[i];
            _balance[_to][_ids[i]] += _values[i];
        }
        emit TransferBatch(msg.sender, _from, _to, _ids, _values);
        require(_checkOnERC1155BatchReceived(msg.sender, _from, _to, _ids,
            _values, _data));
    }
```

Fig. 10: Buggy ERC1155 safeBatchTransferFrom function

samples. The fact that errors that could lead to millionaire losses were detected in real-world contracts attests to the necessity of our framework and its practical impact. We understand that smart contracts is a revolutionary technology and that it attracts more and more attention from new developers, therefore, it is important to make its development process more secure, which will help to give more credibility and spread the technology.

4.3 Threats to Validity and Limitations

Our initial motivation to gather the samples from public github repositories may be a threat to our search strategy. Since we could not analyze private repositories, relevant cases to show strengths and weaknesses of the framework may not have been included. Our approach was successful to identify and analyse nonconformities that went unnoticed during the development process but the relatively low number of samples could also be considered as a threat, since it could lead to an unintentionally biased study or less comprehensive than they could have been.

Our framework also presents some limitations, since it is not in our scope to verify errors on inter-contract interactions. The pre- and postconditions are bound to the methods that make up the contract interface so it is not possible to specify loop invariants.

4.4 Trusted Deployer Tool

After discussing the results of the smart contract verification process, we introduce a scenario based on one of the repositories used during the study to show how our tool would work in practice. In this section, we demonstrate how to integrate all the concepts introduced in this work in a smart contract development process. In addition, we point out the efficiency of our strategy, in comparison to other existing approaches for smart contract upgrade. The trusted deployer is a tool based on our framework, that is, it requires that developers have their code formally verified before they can deploy their contracts to a blockchain network. In order to create or upgrade a smart contract, a developer has to provide its code and specification together. The tool will verify the code against its specification before it proceeds to deploy the smart contract. This tool follows the architecture shown in Figure 3 and relies on five modules: deployer, checker, parser, proxy and database.

Figure 11 presents a reduced verification contract, which is the result of the merging of the specification and implementation contracts. The verification contract is created from the ast tree of the contracts after a syntax check is performed, in this case the goal is to analyze if there is any divergence between the signature of the functions and the data model of the specification and implementation contracts. After this step it is possible to verify all properties with the solc-verify program.

The trusted registry Figure 12 can confirm that a given instance was created by the trusted deployer by calling its *get-spec* function. Since smart contracts

```
contract ERC20Token {

    mapping (address => uint) balances;
    mapping (address => mapping (address => uint)) allowed;
    uint public _totalSupply;

    event Transfer(address indexed _from, address indexed _to, uint _value);
    event Approval(address indexed _owner, address indexed _spender, uint
        _value);

/** @notice postcondition balances[_owner] == balance */
 function balanceOf(address _owner) public view returns (uint balance) {
        return balances[_owner];
    }

/** @notice postcondition allowed[_owner][_spender] == remaining */
function allowance(address _owner, address _spender)  public view returns (
    uint remaining) {
        return allowed[_owner][_spender];
    }
}
```

Fig. 11: Reduced Merged Contract

in a blockchain platform, cannot rely on external services, such as the trusted deployer, that is, they would have no means to check whether an instance that it wants to interact with is safe or not. We create a trusted registry as part of the trusted deployer infrastructure which is essentially a mirror of the trusted deployer's internal registry implemented as a smart contract. It has a maintainer and mapping `verified_addr` that associates the proxy instances that have been created by the trusted deployer with the specification they comply to. As an implementation detail, we do not store the specification themselves but rather a small representative as a 32-byte array - it could be, for instance, a cryptographic hash of the specification. We do not allow this representative to be the zeroed array, $bytes32(0)$ in Solidity syntax, as we use this value to represent absence of an association. That is, if the result of a call $get_spec(addr)$ is the value $bytes32(0)$, it means the address $addr$ has not been deployed by the trusted deployed, and hence has no specification associated with it.

The proxy pattern as discussed in Section 3.2 separates the logic and the data of a contract. The key concept to understand is that the logic contract can be replaced while the implemented trusted proxy Figure 13, or the access point is never changed. Both contracts are still immutable in the sense that their code cannot be changed, but the logic contract can simply be swapped by another contract. If an error occurs during the process execution it is rolled back and an error message will be returned to the user otherwise the contract will send the contract to the blockchain. The results collected from our evolution scenario, one can see that our strategy is effective. The error contained in the commit 548fda was identified and the developer would be forced to fix it. Our tool abstracts many details of the deploy and upgrade process making it faster when compared to the manual process.

```
contract Registry {
    address maintainer;
    mapping (address => bytes32) verified_addrs;

    constructor() public {
        maintainer = msg.sender;
    }
    function new_mapping(address addr, bytes32 spec_id) public {
        if (msg.sender == maintainer && spec_id != bytes32(0)) {
            verified_addrs[addr] = spec_id;
        }
    }
    function get_spec(address addr) view public returns (bytes32) {
        return verified_addrs[addr];
    }
}
```

Fig. 12: Trusted registry smart contract

```
contract Proxy {

    mapping(address => uint256) internal balances;
    mapping(address => mapping(address => uint256)) internal allowed;
    uint256 public _totalSupply;

    function balanceOf (address _owner) public returns (uint256 balance) {
        (bool success, bytes memory bytesAnswer) = implementation.
            delegatecall( abi.encodeWithSignature("balanceOf(address)",
            _owner));
        require(success);
        return abi.decode(bytesAnswer, (uint256));
    }
    function allowance (address _owner,address _spender) public   returns (
         uint256 remaining) {
        (bool success, bytes memory bytesAnswer) = implementation.
            delegatecall(abi.encodeWithSignature("allowance(address,address)"
            , _owner,_spender));
        require(success);
        return abi.decode(bytesAnswer, (uint256));
    }
}
```

Fig. 13: Implemented Trusted Proxy

5 Related Work

There is a glaring need for a safe mechanism to upgrade smart contracts in platforms, such as Ethereum, where contract implementations are immutable once deployed; the many surveys uncovering this fact [19,40,15] and community-proposed design patterns proposing mechanisms to upgrade smart contracts [38,10,25,32] attest this necessity. Yet, surprisingly, we could only find two close related approaches [11,35] that try to tackle this problem. The preliminary work in [11] proposes a methodology based around special contracts that carry a proof that they meet the expected specification. Their on-chain solution requires fundamental changes in the smart contract platforms themselves. They propose the

addition of a special instruction to deploy these special proof-carrying contracts, and the adaptation of platform miners, which are responsible for checking and reaching a consensus on the validity of contract executions, to check these proofs. Our framework and the one presented in that work share the same goal: to propose a mechanism by which contracts can be upgraded but only if they meet the expected specification. However, our approach and theirs differ significantly in many aspects. Firstly, while theirs requires a fundamental change on the rules of the platform - which requires a large distributed network of nodes, i.e. the smart contract platform, to agree upon - ours can be implemented, as already prototyped, on top of Ethereum's current capabilities and can rely on tools that are easier to use, i.e. require less user input, like program verifiers. The fact that their framework is on-chain makes the use of such verification methods more difficult since these methods would slow down consensus, likely to a prohibitive level. Finally, while they introduce abstract ideas with some concrete elements, we provide details on how to implement our framework using current technology and an evaluation based on real-world Solidity samples.

In [35], the authors propose a mechanism to upgrade contracts in Ethereum that works at the EVM-bytecode level. Their framework takes vulnerability reports issued by the community as an input, and tries to patch affected deployed contracts automatically using patch templates. It uses previous contract transactions and, optionally user-provided unit tests, to try to establish whether a patch preserves the behaviour of the contract. Ultimately, the patching process may require some manual input. If the deployed contract and the patch disagree on some test, the user must examine this discrepancy and rule on what should be done. Note that this manual intervention is always needed for attacked contracts, as the transaction carrying out the attack - part of the attacked contract's history - should be prevented from happening in the new patched contract. While they simply test patches that are reactively generated based on vulnerability reports, we proactively require the user to provide a specification of the expected behaviour of a contract and formally verify the evolved contract against such a formal specification. Their approach requires less human intervention, as a specification does not need to be provided - only optionally some unit tests - but it offers no formal guarantees about patches. It could be that a patch passes their validation (i.e. testing with the contract history), without addressing the underlying vulnerability.

Some other papers have proposed methodologies to carry out pre-deployment patching/repairing [41,31,47]. They try to scan a binary for common vulnerabilities and patch the vulnerabilities they find prior to deploying the contract. These papers do not propose a way to update deployed contracts.

A number of analysis tools for EVM bytecode were designed to find specific behaviour patterns witnessing typical bad behaviours [26,30,24,14,39,42,33]. Tools operating on the level of Solidity were also proposed [45,17,16,7,5,4]. These tools tend to focus instead on formally verifying user-provided semantic properties. Our paper proposes a verification-focused development process based around, supported, and enforced by such tools.

Design by contract [28] is a methodology that was originally created for specifying the behaviour of object-oriented programs but was also adopted in other contexts [29,22,9,21,20,17]. This sort of specification is particularly fitting in the case of Solidity smart contracts, especially the format of specification that we propose, as the community already employ a similar format, albeit informal, to describe standard contract interfaces in the form of Ethereum Request for Comments (ERCs); see for example, ERC20 [43].

6 Conclusion

We propose a framework for the safe deployment of smart contracts. Not only does it check that contracts conform to their specification at creation time, but it also guarantees that subsequent code updates are conforming too. Upgrades can be performed even if the implementation has been proven to satisfy the specification initially. A developer might, for instance, want to optimise the resources used by the contract. Furthermore, our *trusted deployer* records information about the contracts that have been verified, and which specification they conform to, so that participants can be certain they are interacting with a contract with the expected behaviour; contracts can be safely executed. None of these capabilities are offered by the Ethereum platform by default nor are available in the literature to the extent provided by the framework proposed in this paper.

We have prototyped our trusted deployer and investigated its applicability - specially its formal verification component - to contracts implementing two widely used Ethereum standards: the ERC20 Token Standard and ERC1155 Multi Token Standard, with promising results.

This idea of using trusted computing to verify a smart contract before its deployment can be extended to software in general. Particularly, a trusted deployer could be part of a deployment process for reactive systems in general, such as component-based, (micro)service-based systems, or even system-of-systems.

Our framework shifts immutability from the implementation of a contract to its specification, promoting the "code is law" to the "specification is law" paradigm. We believe that this paradigm shift brings a series of improvements. Firstly, developers are required to outline their intent in the form of a (formal) specification, so they can, early in the development process, identify issues with their design. They can and should validate their specification; we consider this problem orthogonal to the framework that we are providing. Secondly, specifications are more abstract and, as a consequence, tend to be more stable than (the corresponding conforming) implementations. A contract can be optimised, for instance, and both the original and optimised versions must satisfy the same reference specification. Thirdly, even new implementations that involve change of data representation can still be formally verified against the same specification, by using data refinement techniques.

A limitation of our current approach is the restrictive notion of evolution for smart contracts: only the implementation of public functions can be upgraded - the persistent state data structures are fixed. However, we are looking into new

types of evolution where the data structure of the contract's persistent state can be changed - as well as the interface of the specification, provided the projected behaviour with respect to the original interface is preserved, based on notions of class [23] and process [12] inheritance, and interface evolution such as in [13].

References

1. Ethereum White Paper. `https://github.com/ethereum/wiki/wiki/White-Paper`.
2. Ethereum Yellow Paper. `https://ethereum.github.io/yellowpaper/paper.pdf`.
3. *Solidity compiler*. `https://github.com/ethereum/solidity`.
4. Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 9–24, Cham, 2020. Springer International Publishing.
5. Wolfgang Ahrendt, Richard Bubel, Joshua Ellul, Gordon J. Pace, Raúl Pardo, Vincent Rebiscoul, and Gerardo Schneider. Verification of smart contract business logic. In Hossein Hojjat and Mieke Massink, editors, *Fundamentals of Software Engineering*, pages 228–243, Cham, 2019. Springer International Publishing.
6. Pedro Antonino and A. W. Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity, 2020.
7. Pedro Antonino and A. W. Roscoe. Solidifier: Bounded model checking solidity using lazy contract deployment and precise memory modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1788–1797, 2021.
8. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST 2017*, pages 164–186. Springer, 2017.
9. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, pages 364–387. Springer, 2005.
10. Gabriel Barros and Patrick Gallagher. EIP-1822: Universal Upgradeable Proxy Standard (UUPS). `https://eips.ethereum.org/EIPS/eip-1822`.
11. Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-carrying smart contracts. In *Financial Cryptography Workshops*, 2018.
12. José Dihego, Pedro R. G. Antonino, and Augusto Sampaio. Algebraic laws for process subtyping. In Lindsay Groves and Jing Sun, editors, *Formal Methods and Software Engineering - 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1, 2013, Proceedings*, volume 8144 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2013.
13. José Dihego, Augusto Sampaio, and Marcel Oliveira. A refinement checking based strategy for component-based systems evolution. *J. Syst. Softw.*, 167:110598, 2020.
14. Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep*, 2018.
15. Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 634–653, Cham, 2020. Springer International Publishing.

16. Ákos Hajdu and Dejan Jovanović. Smt-friendly formalization of the solidity memory model. In *ESOP 2020*, pages 224–250. Springer, 2020.
17. Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *VSTTE*, pages 161–179. Springer, 2020.
18. Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *CSF 2018*, pages 204–217. IEEE, 2018.
19. Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. *Patterns*, 2(2):100179, 2021.
20. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*, pages 175–188. Springer US, Boston, MA, 1999.
21. K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.
22. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
23. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.
24. Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *ICSE 2018*, pages 65–68. ACM, 2018.
25. Alan Lu. Solidity DelegateProxy Contracts. `https://blog.gnosis.pm/solidity-delegateproxy-contracts-e09957d0f201`.
26. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS 2016*, pages 254–269. ACM, 2016.
27. P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.
28. B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.
29. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., USA, 1st edition, 1988.
30. Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.
31. Tai D. Nguyen, Long H. Pham, and Jun Sun. Sguard: Towards fixing vulnerable smart contracts automatically. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1215–1229, 2021.
32. Santiago Palladino. EIP-1967: Standard Proxy Storage Slots. `https://eips.ethereum.org/EIPS/eip-1967`.
33. Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *S&P 2020*, pages 18–20, 2020.
34. Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. EIP-1155: Token Standard. `https://eips.ethereum.org/EIPS/eip-1155`.

35. Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Evmpatch: Timely and automated patching of ethereum smart contracts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1289–1306. USENIX Association, August 2021.
36. AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. 2020.
37. David Siegel. Understanding the dao attack. `https://www.coindesk.com/understanding-dao-hack-journalists` accessed on 22 July 2021.
38. OpenZeppelin team. Proxy Upgrade Pattern. `https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies`.
39. Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *WETSEB 2018*, pages 9–16. IEEE, 2018.
40. Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. 54(7), 2021.
41. Christof Ferreira Torres, Hugo Jonker, and Radu State. Elysium: Automagically healing vulnerable smart contracts using context-aware patching. *CoRR*, abs/2108.10071, 2021.
42. Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *CCS 2018*, pages 67–82. ACM, 2018.
43. Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard. `https://eips.ethereum.org/EIPS/eip-20`.
44. Jan Vollmer. The biggest hacker whodunnit of the summer. `https://www.vice.com/en/article/pgkzqm/the-biggest-hacker-whodunnit-of-the-summer` accessed on 22 July 2021.
45. Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *VSTTE*, pages 87–106, 2020.
46. Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. Ace: Asynchronous and concurrent execution of complex smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 587–600, 2020.
47. Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, 29(4), September 2020.