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 three widely used Ethereum standards: the ERC20 Token Standard, ERC3156 Flash Loans 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 [30,7]. 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* [31] 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. Therefore, 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* [25]. 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 [35]. 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. 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 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, ERC3156 and ERC1155 Ethereum token standards. 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) [24], specifically the AMD SEV implementation [29]. Despite being an off-chain service, the use of a TEE to implement our deployer should give it a level of *execution integrity/trustworthiness*, enforced by the trusted hardware, comparable to that achieved by on-chain mechanisms relying on blockchains' *consensus*, with less computational overhead. However, on-chain mechanisms would enjoy better availability guarantees. We further discuss these trade-offs in Section 5.
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 Solidity

A smart contract is a program running on a trusted platform, usually a blockchain, that manages the digital assets it owns. Solidity is arguably the most used language for writing smart contracts as it was designed for the development of contracts targeting the popular Ethereum blockchain platform [1]. 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 (Ethereum's cryptocurrency). 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 *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., $\sigma'[\texttt{addr}_{\texttt{tw}}].balance = 0$. 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 [18,36,6].

A transaction *call-contract* can be used to invoke contract functions; processing *call-contract*$(addr, func\_sig, args)$ executes the function with signature

```
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.

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 [9,20]. 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 preconditions, any state 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(val)("")` instead of `msg.sender.send(val)`. While the latter only allows for the trans-

```
/** @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.

fer of `val` 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 [30,7].

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.

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 [37]. 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

[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 side-effect computation. So, for the purpose of this paper and to simplify our exposition, we ignore this delegation.

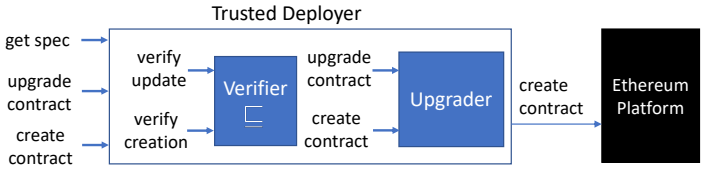

Fig. 3: Trusted deployer architecture.

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* (*upgrade-contract*) checks 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 - 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* [25] 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 preconditions are guaranteed by the caller code and the method terminates. Currently, we focus on partial correctness, which is aligned with our goal to ensure safety properties, and the fact that smart contracts typically have explicitly bound executions[6]. 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. Preconditions 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* specification language, which in turn borrows elements from the Boogie language [9,20]. 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 [6] 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.

[6] The Ethereum concept of *gas*, i.e. execution resources, is purposely abstracted away/disregarded in our exposition.

```
/** @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.

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 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 [17,5,36]. 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* [31], 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`.[7] This proxy contract also has a public function `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 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.

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

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 Case studies: ERC20, ERC1155, and ERC3156

To validate our approach, we have carried out three systematic case studies of the ERC20 Token Standard, the ERC1155 Multi Token Standard, and the ERC3156 Flash Loans. For the ERC20, we examined 8 repositories and out of 32 commits analysed, our framework identified 8 *unsafe commits*, in the sense that they did not conform to the specification; for the ERC1155, we examined 4 repositories and out of 18 commits analysed, 5 were identified as unsafe; and for the ERC3156, we examined 5 repositories and out of 18 commits analysed, 7 were identified as unsafe. We have prototyped the entire framework in the form of our Trusted Deployer.[8] We have applied it to the commit history of the repository $0xMonorepo$, and our tool was able to identify and prevent unsafe evolutions while carrying out safe ones. The design and promising findings of these case studies and commit history analyses are presented in full detail in the extended version of this paper [4]. In the remainder of this section, as our space is limited, we only present here a brief account of the ERC20 case study.

Our summary of the ERC20 case study presented here has focused specifically on the verification of the semantic obligation that we enforce. This task is the most important and computationally-demanding element of our methodology. So, in this case study, we try to establish whether: (a) can we use our notation to capture the ERC20 specification formally, (b) (if (a) holds) can *solc-verify* check that real-world ERC20 contracts conform to its formal specification, and (c) (if (b) holds) how long does *solc-verify* take to carry out such verifications.s

We were able to capture the ERC20 specification using our notation, an extract of which is presented in Figure 5, so we have a positive answer to (a). To test (b) and (c), we relied on checking, using *solc-verify*, merged contracts involving our specification and real-world contracts. We selected contract samples from public github repositories that presented a reasonably complex and interesting commit history. The samples 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.

[8] The prototype is implemented as a standalone tool available at `https://github.com/formalblocks/safeevolution`. We do not provide a service running inside a Trusted Execution Environment yet but such a service will be provided in the future.

ERC20							
Repository	Commit	Time	Output	Repository	Commit	Time	Output
0xMonorepo	548fda	7.78s	WOP	Uniswap	55ae25	6.89s	WOP
DigixDao	5aee64	8.52s	NTI	Uniswap	e382d7	7.08s	IOU
DsToken	08412f	8.74s	WOP	SkinCoin	25db99	1.95s	NTI
Klenergy	60263d	2.40s	VRE	SkinCoin	27c298	1.81s	NTI

Table 1: ERC20 Results

```
1  /** @notice invariant totalSupply == __verifier_sum_uint(balances) */
2  contract IERC20  {
3  uint256 totalSupply;
4  mapping (address => uint256) balances;
5  mapping (address => mapping (address => uint256)) allowance;
6
7  //... functions transfer, totalSupply, balanceOf, allowance, and approve
       omitted ...
8
9  /** @notice postcondition ((balances[from] == __verifier_old_uint(balances[
       from]) - value && from != to) || (balances[from] == __verifier_old_uint(
       balances[from]) && from == to) && ok) || !ok
10 * @notice postcondition ((balances[to] == __verifier_old_uint(balances[to]) +
        value && from != to) || (balances[to] == __verifier_old_uint (balances[
       to]) && from == to) && ok) || !ok
11 * @notice postcondition allowance[from][msg.sender] == __verifier_old_uint(
       allowance[from][msg.sender]) - value || from == msg.sender
12 * @notice postcondition allowance[from][msg.sender] <= __verifier_old_uint(
       allowance[from][msg.sender]) || from == msg.sender
13 * @notice postcondition forall (address addr) (addr == from || addr == to ||
       __verifier_old_uint(balances[addr]) == balances[addr]) && ok || (
       __verifier_old_uint(balances[addr]) == balances[addr]) && !ok*/
14 function transferFrom(address from, address to, uint256 value) public returns
        (bool ok);
15 }
```

Fig. 5: ERC20 reduced specification.

We checked these merged contracts using a Lenovo IdeapadGaming3i with 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. Table 1 shows the results we obtained.[9] Our framework was able to identify errors in 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; Verification Error (VRE) denotes that the verification process cannot be completed or the results were inconclusive. Our framework also found conformance for 24 commits analysed; we omitted those for brevity, each of them was verified in under 10s.

The ERC20 standard 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.

[9] All the instructions, the specifications, the sample contracts, and scripts used in this evaluation can be found at `https://github.com/formalblocks/safeevolution`.

```
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.

Figure 5 presents a reduced specification - focusing on function `transferFrom` for the purpose of this discussion - derived from the informal description in the standard [35]. In Line 1, we define a contract invariant requiring that the total number of tokens supplied by this contract is equal to the sum of all tokens owned by account holders. 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 9 to 10) require that the balances are updated as expected, whereas the purpose of the last two (lines 11 to 12) 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. We have also validated cases of safe evolution, namely, where our framework was able to show that consecutive updates conformed with the specification.

The results of our case study demonstrate that we can verify real-world contracts implementing a very popular Ethereum token standard efficiently - positively answering questions (b) and (c). The fact that errors were detected (and safe evolutions were checked) in real-world contracts attests to the necessity of our framework and its practical impact. More details about this case study and of the other two, with our commit history analyses, can be found in [4].

5 Related Work

Despite the glaring need for a safe mechanism to upgrade smart contracts in platforms, such as Ethereum, where contract implementations are immutable once deployed [19,32,15], surprisingly, we could only find three close related approaches [10,8,28] that try to tackle this specific problem. The work in [10] proposes a methodology based around special contracts that carry a proof that they meet the expected specification. 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, but our approach and theirs differ significantly in many aspects. Firstly, while theirs requires a fundamental change on the rules of the platform, ours can be implemented, as already prototyped, on top of Ethereum's current capabilities and 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 as they would slow down consensus, likely to a prohibitive level.

Azzopardi *et al.* [8] propose the use of runtime verification to ensure that a contract conforms to its specification. Given a Solidity smart contract C and an automaton-based specification S, their approach produces an instrumented contract I that dynamically tracks the behaviour of C with respect to S. I's behaviour is functionally equivalent to C when S is respected. If a violation to S is detected, however, a reparation strategy (i.e. some user-provided code) is executed instead. This technique can be combined with a proxy to ensure that a monitor contract keeps track of implementation contracts as they are upgraded, ensuring their safe evolution. Unlike our approach, there is an inherent (on-chain) runtime overhead to dynamically keep track of specification conformance. An evaluation in that paper demonstrates that, for a popular type of contract call, it can add a 100% cost overhead. Our off-chain verification at deployment-time does not incur this sort of overhead. Another difference from our approach concerns the use of reparation strategies. One example given in the paper proposes the reverting of a transaction/behaviour that is found to be a violation. An improper implementation could, then, have most of its executions reverted. Our approach presents at (pre-)deployment-time the possible violated conditions, allowing developers to fix the contract before deployment. Their on-chain verification can be implemented on top of Ethereum's capabilities.

In [28], 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.

Methodologies to carry out pre-deployment patching/repairing of smart contracts have been proposed [33,26,38]. However, they do not propose a way to update deployed contracts. A number of tools to verify smart contracts at both EVM and Solidity levels have been proposed [23,22,14,13,34,27,36,17,16,6,2,3]. Our paper proposes a verification-focused development process based around, supported, and enforced by such tools.

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 three widely used Ethereum standards: the ERC20 Token Standard, ERC3156 Flash Loans and ERC1155 Multi Token Standard, with promising results.

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 elaborate 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 so that 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 [21] and process [11] inheritance, and interface evolution such as in [12].

References

1. Ethereum White Paper. `https://github.com/ethereum/wiki/wiki/White-Paper` accessed on 5 August 2022.
2. Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In *ISoLA '20*, pages 9–24. Springer, 2020.
3. Leonardo Alt and Christian Reitwiessner. Smt-based verification of solidity smart contracts. In *ISoLA 2018*, pages 376–388. Springer, 2018.
4. Pedro Antonino, Juliandson Ferreira, Augusto Sampaion, and A.W. Roscoe. Specification is law: Safe deployment of ethereum smart contracts - technical report. Tech. report, 2022. `https://github.com/formalblocks/safeevolution`.
5. Pedro Antonino and A. W. Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity. *CoRR*, abs/2002.02710, 2020.
6. Pedro Antonino and A. W. Roscoe. Solidifier: Bounded model checking solidity using lazy contract deployment and precise memory modelling. In *SAC '21*, page 1788–1797, 2021.
7. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST 2017*, pages 164–186. Springer, 2017.
8. Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 113–137. Springer, 2018.
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. Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-carrying smart contracts. In *Financial Cryptography Workshops*, 2018.
11. José Dihego, Pedro R. G. Antonino, and Augusto Sampaio. Algebraic laws for process subtyping. In Lindsay Groves and Jing Sun, editors, *ICFEM 2013*, volume 8144 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2013.
12. José Dihego, Augusto Sampaio, and Marcel Oliveira. A refinement checking based strategy for component-based systems evolution. *J. Syst. Softw.*, 167:110598, 2020.
13. Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774. USENIX Association, August 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 *FC 2020*, pages 634–653, Cham, 2020. Springer.
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. K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.
21. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.
22. 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.
23. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS 2016*, pages 254–269. ACM, 2016.
24. 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.
25. B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.
26. 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.
27. 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.
28. Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Evmpatch: Timely and automated patching of ethereum smart contracts. In *USENIX Security 21)*, pages 1289–1306. USENIX Association, August 2021.
29. AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. 2020.
30. David Siegel. Understanding the dao attack. `https://www.coindesk.com/understanding-dao-hack-journalists` accessed on 22 July 2021.
31. OpenZeppelin team. Proxy Upgrade Pattern. `https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies` accessed on 5 August 2022.
32. Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Comput. Surv.*, 54(7), 2021.
33. Christof Ferreira Torres, Hugo Jonker, and Radu State. Elysium: Automagically healing vulnerable smart contracts using context-aware patching. *CoRR*, abs/2108.10071, 2021.
34. 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.
35. Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard. `https://eips.ethereum.org/EIPS/eip-20` accessed on 5 August 2022.
36. 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.
37. Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. Ace: Asynchronous and concurrent execution of complex smart contracts. In *CCS '20*, 2020.
38. Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, 29(4), September 2020.