A refinement-based approach to safe smart contract deployment and evolution

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

[1]The Blockhouse Technology Limited, Oxford, UK.
[2]Centro de Informática, Universidade Federal de Pernambuco, Recife, PE, Brazil.
[3]Department of Computer Science, Oxford University, Oxford,UK.
[4]University College Oxford Blockchain Research Centre, Oxford,UK.

Contributing authors: pedro@tbtl.com; jef@cin.ufpe.br; acas@cin.ufpe.br; awroscoe@gmail.com; fmca@cin.ufpe.br;

Abstract

In our previous work, we proposed a verification framework that shifts from the "code is law" to a new "specification is law" paradigm related to the safe evolution of smart contracts. The framework proposed there relaxed the well-established requirement that, once a smart contract is deployed in a blockchain, its code is expected to be immutable. More flexibly, contracts are allowed to be created and upgraded provided they meet a corresponding formal specification that was fixed. In the current paper, we extend this framework to allow specifications to evolve, provided a refinement notion is preserved. We propose a notion of specification refinement tailored for smart contracts and a methodology for checking it. In addition to weakening preconditions and strengthening postconditions and invariants, we allow for the change of data representation and interface extension. Thus, we are able to reason about a significantly wider class of smart contract evolution histories, when contrasted with the original framework. The new framework is centered around *a trusted deployer*: an off-chain service that formally verifies and enforces the notions of implementation conformance and specification refinement. We have investigated its applicability to the safe deployment and upgrade of contracts implementing widely used Ethereum standards (the ERC20 Token Standard, the ERC3156 Flash Loans, the ERC1155 Multi Token Standard and The ERC721 standard for Non-Fungible Tokens); we handle evolutions possibly involving changes in data representation and interface extensions.

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

1 Introduction

In the context of trusted computing, smart contracts were proposed as a way to extended the capabilities of blockchains. Blockchains were initially created as a distributed and decentralised record for transactions capturing the transfer of digital currency/assets. So, their transaction processing logic was fixed and primarily focused on preventing *double spending*: the ability of using the same digital coin in two separate valid transactions. With the advent of smart contracts, blockchain users have the flexibility of using a program to define how a transaction is processed.

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. The use of a trusted platform is required to guarantee that it is computationally infeasible to tamper with its execution or persistent state in an undetectable way.

In this perspective, smart contracts 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 [63, 71, 13]. 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* [65] as a mechanism by which one can mimic contract upgrades by splitting a contract into two instances: the *proxy instance* holds the contract's persistent state, and the *implementation instance* the code associated with its functions. The proxy instance stores a pointer to the current implementation instance, and relies on the code deployed on the latter to execute. Hence, an update can be carried out (i.e. mimicked) by: (a) deploying a new implementation instance, and (b) changing the proxy instance to point to this new instance. To avoid arbitrary updates, a specific platform participant, the maintainer of the contract, is given solo access to performing (b) and is expected to perform reasonable updates.

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* [48]. 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 [70].

Ethereum has already a mechanism by which the community can propose and agree on smart contract specifications (amongst other artifacts), called Ethereum Request for Comments (ERCs) — see ERC20 [70], for instance. These specifications describe the function signatures of a compliant contract implementation together with a brief textual, informal, explanation on how the functions should behave. The format that we propose for our specifications is very similar to that, except that we use a formal notation to capture the behaviour of public functions. Our framework also relies on our own version of the proxy diamond pattern [52] 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 implementation and specification creations and updates. 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.

In previous work, we proposed a framework that promoted a paradigm shift where the specification is immutable instead of the implementation/code [10]. Thus, it moves away from "code is law" and proposes the *"specification is law" paradigm* — enforced by formal verification. In this context, an initial deployment or subsequent upgrade is allowed only if the given implementation conforms to the associated specification, which is fixed for a given contract evolution history. Conformance of an implementation to a specification is performed in a compositional way, by checking whether each public function obeys its specified properties described as design contracts.

The main contribution of the current paper is to extend the framework presented in [10] to handle the verification of a larger class of smart contract evolution. Particularly, instead of fixing the specification of a smart contract at creation time, it allows the specification to evolve as long as the new specification preserves a refinement notion tailored for smart contract evolution. Unlike the previous framework, the one proposed here copes with change of data representation and interface extension, and allows preconditions to be weakened and postconditions and invariants to be strengthened. This ensures that, for instance, properties that are derived from the invariant of the initial *reference* specification of a smart contract have to be enforced by any updated specification, thanks to our refinement relation. Also, concerning interface extensions, new public functions are obliged to preserve existing contract invariants. In summary, a safe deployment or upgrade in the context of the extended framework imposes both an implementation conformance obligation, as in the framework proposed in [10], and also a specification refinement verification in situations where the specification evolves.

This safe evolution 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. Furthermore, specifications can evolve, as already mentioned, but only in a controlled way as dictated by a refinement notion we propose.

We have encoded the specification refinement verifications using the *solc-verify* tool [32, 31], considering all the relevant artifacts. We then conducted a case study that investigates the applicability of the overall approach to real-world smart contracts implementing the widely used ERC20 [70], ERC3156 [20], ERC1155 [59] and ERC721 [25] Ethereum token standards. We analysed specifically how the sort of formal verification that we use fares in handling practical contracts and obtained promising results. Whereas in [10] we focused on evolution scenarios with a fixed specification, in the case study developed here we explore evolution histories that embody different data representations and interface extensions.

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) [46], specifically the AMD SEV implementation [61], using a protocol like [9].

We present an infrastructure that applies to the Ethereum platform and for contracts written in Solidity. However, the ideas proposed here should be applicable to similar platform and to Ethereum contracts written in languages other than Solidity. They could even be applied to other types of systems such as component-based, (micro)service-based, or system-of-systems [54, 57, 33].

In summary, this paper significantly improves and extends [10] in the following ways.

- We support specification evolution, embodying both the possibility of change of data representation and interface extension. This allows us to verify the safe deployment and upgrade of smart contracts that our previous framework could not handle.
- In addition to the previous conformance notion between an implementation and a specification,

we propose a notion of specification refinement and extend our framework to incorporate this ability, particularly proposing a methodology to check specification refinement using the *solc-verify* prover. Section 3.1.1 is entirely new.
- We capture and formalise a notion of (persistent) state transformation to allow contracts to safely change their state representation dynamically. Section 3.1.3 is also new.
- We demonstrate a number of (meta-)properties that are enforced by our framework via its safe deployment and upgrade mechanisms in Sections 3.1.1 and 3.1.2.
- We illustrate the practical applicability of the extended framework on contracts implementing widely used Ethereum token standards, particularly considering evolutions involving change of data representation and interface extension. Section 4.2 is new.
- We have also extended the sections on background, case studies (Section 4.3 is new), and related work.

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

We briefly discuss some relevant aspects of the Ethereum blockchain and the Solidity language, and then introduce some verification features of the *solc-verify* tool.

2.1 Blockchain and smart contracts

Blockchain is a chronologically ordered chain of blocks, also referred to as a distributed ledger managed in a decentralised manner. The chaining is done by adding the cryptographic hash of the previous block to the current block. This mechanism makes it infeasible to tamper with the data stored, since a transaction cannot be changed without changing its block and all the successor blocks [8]. Furthermore, for any attack to succeed, an adversary must take control over half of the whole network computational power. It is usually referred to as a "shared" or "distributed" ledger since all network participants have a copy and share responsibility for keeping it up to date. This is the main difference between blockchain and centralised systems, where control is normally maintained by an institution [77]. The transactions contained therein are transparent, auditable, reliable, anonymous and there is no need for intermediaries. The decentralised nature of a blockchain motivates attackers to make fraudulent transactions or even reverse legitimate ones, which can create competing or inaccurate ledgers. At its core blockchains are replicated state machines (RSMs) capable of transitioning to a new state based on external interactions and documenting previous states under a Byzantine fault tolerance model [39]. An important property of blockchains is that there must be a single valid state, which is accessible to anyone connected to the network. In this sense, a blockchain can be thought of as a kind of distributed database with a single shared state, which can be updated via the addition of blocks of transactions [62]. Although it was initially presented as a solution to specific problems related to the financial services industry, blockchain technology can be adapted to other industries where information integrity is necessary. Examples of use cases at various stages of maturity are: auctions [30], data management systems [5], financial contracts [18], elections [47] and trading platforms [55].

Smart contracts are programs stored on a blockchain that automatically enforce their terms when predetermined conditions are met [79]. Furthermore, similarly to traditional contracts, smart contracts may define rules and penalties around an agreement, but they perform the obligations automatically, significantly improving four basic aspects of contracts: observability, verifiability, privacy, and enforceability.

2.2 Ethereum and Solidity

Ethereum is an open source, decentralised, distributed computing platform that provides a virtual machine capable of executing smart contracts using blockchain technology. It was created to make life easier for developers who want to create decentralised applications [74]. Unlike the Bitcoin network, it was designed to support a wide range of industries although currently most applications are geared towards the financial sector.

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 the 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. In this paper, we focus on Solidity versions 0.5.* and 0.6.*. Although, our ideas could be implemented for other versions too.

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

```
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.

```
/** @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.

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., $\sigma'[addr_{\texttt{tw}}].balance = 0$. 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 [35, 73, 12].

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 $\texttt{addr}_{tw}$.

2.3 Formal verification with *solc-verify*

Formal verification embodies the application of rigorous mathematical methods to reason about the correctness of a system with respect to a certain formal specification or property. Many tools have emerged to support the development and analysis of smart contracts.

The modular verifier *solc-verify* [32, 31] 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 [15, 40]. 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 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.

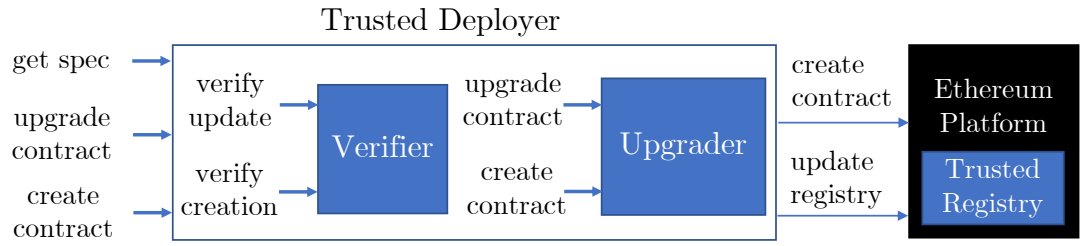

Fig. 3: Trusted deployer architecture.

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`.[1] 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 [63, 71, 13].

3 Safe Ethereum Smart Contract Deployment and Upgrade

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.

Besides a conformance relation between implementations and specifications, we define a refinement notion between specifications. This allows a specification to evolve as well, provided this notion of refinement is preserved. Apart from weakening preconditions and strengthening postconditions and invariants, specification evolution allows both interface extension and changing data representation of smart contracts. We rely on traditional notions of program refinement [50] and behavioural subtyping [42] to propose our notion of specification refinement.

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 [75]. 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 and a summary of its functions in Table 1. The trusted deployer relies on an internal *verifier* that implements the functions *verify-creation* and *verify-upgrade*, and an *upgrader* that implements functions *create-contract* and *upgrade-contract*.

The deployer's *create-contract* function receives as input a specification S and an implementation C; it checks whether C conforms to S by calling *verify-creation*, before relaying this call to the upgrader's *create-contract* that effectively creates the upgradable contract in the Ethereum

[1] 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.

S	Contract specification.
C	Contract implementation.
α	Abstraction function.
`init`	Initialisation function used for state migration.
addr	Address of deployment (and contract identifier).
get_spec(*addr*) : S	Trusted deployer function returning the current specification for the contract at address *addr*.
create_contract(S, C) : *addr*	Trusted deployer function that checks the conformance of C to S and create the upgradable contract using C, returning the address *addr* of deployment (i.e. the proxy address).
upgrade_contract(*addr*, S, C, α, `init`)	Trusted deployer function that checks the conformance of C to S, that S refines $\bar{S}$ (i.e. the current specification for contract *addr*) given abstraction function α. It also upgrades *addr* using C, migrating its state using `init`.
verify_creation(S, C)	Verifier function that checks the conformance of implementation C to specification S.
verify_upgrade(*addr*, S, C, α, `init`)	Verifier function that checks the conformance of S to C, that S refines $\bar{S}$ (i.e. the current specification for contract *addr*) given abstraction function α, and that `init` is a valid migration function.
create_contract(C) : *addr*	Upgrader function that creates an upgradate contract infrastructure using C. It returns the address *addr* of deployment (i.e. the proxy address).
upgrade_contract(*addr*, C, `init`)	Upgrader function that upgrades the contract at address *addr* using initialisation function `init`.

Table 1: Trusted deployer functions and parameters summary.

platform, and stores the pair (S, C) of specification and implementation currently associated with an upgradable smart contract. This function returns the deployed address *addr* (i.e. the address of the deployed proxy contract as we explain later). This address is used to identify contract instances managed by the trusted deployer.

The *upgrade-contract* function receives as input a specification S, an implementation C, an abstraction function α, and an `init` function, in addition to the address *addr* of the instance to be upgraded. It checks whether C conforms to S and whether S refines the current specification for *addr* by calling *verify-upgrade*, before relaying this call to the upgrader's *upgrade-contract* that effectively upgrades the contract in the Ethereum platform, and records the pair (S, C) as the current specification and implementation associated with the smart contract *addr*. The *get-spec* function can be used to check whether a contract instance has been deployed by the trusted deployer and which specification it currently satisfies. It can be used, for instance, to implement guarded calls where an external service interacting with Ethereum smart contracts can check whether a given contract implements the expected specification before calling it. Only the creator (i.e. *designated maintainer*) of such a *trusted contract* can update its implementation and associated specification. We expand on the creation and update processes and the role of the verifier and upgrader next.

Note that once a contract is created via our trusted deployer, all upgrades are guaranteed to include a refined specification and a conforming implementation. Therefore, participants in the ecosystem interacting with this contract instance can predict what its behaviour might be during the instance's entire lifetime, even if the specification and implementation evolve.

3.1 Verifier

We propose *design-by-contract* [48] 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; we do not address termination. 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 pre- and postconditions, and the specification with invariants; these annotations capture the expected behaviour of the contract.

The verifier implements two functions. Function *verify-creation* takes a specification S and a contract implementation C and tests whether C conforms to S. Function *verify-upgrade* takes a specification S, a contract implementation C, an abstraction function α, and a `init` Solidity function. It checks whether C meets S, as for *verify-creation*, but it also checks whether S refines the currently stored specification by the framework, say $\bar{S}$, and whether `init` is a sound state-transformation function. We detail what these proof obligations — *specification refinement*, *implementation conformance*, and *state-transformation soundness* — entail and how they are carried out in the following.

When the verifier shows that a proof obligation does not hold, it yields an error report that points out possible flaws in a specification/implementation and maybe even witnesses/counterexamples describing system behaviours illustrating such violations; they provide valuable information to help developers correct their evolved implementations/specifications.

3.1.1 Specification refinement

In [10], we have allowed for the code of a smart contract to be updated while its specification remained fixed. As a new contribution here, we allow specifications to evolve provided they obey a refinement relation. The proposed notion of specification refinement is inspired by traditional notions of program refinement [50] and behavioural subtyping [42].

We rely on an abstract definition of *specification.*[2]

Definition 1 A specification is a triple (D, I, A) such that:

- The *data representation* $D = \langle (n_1, t_1), \ldots, (n_{|D|}, t_{|D|}) \rangle$ is a finite sequence that defines the member variables of the specification, each of which with its unique name n_i and associated type t_i.
- The *interface* I is a finite mapping from function names to their signatures. A signature is given by a pair of sequences $(ins = \langle (in_1, it_1), \ldots, (in_{|ins|}, it_{|ins|}) \rangle, outs = \langle (out_1, ot_1), \ldots, (out_{|outs|}, ot_{|outs|}) \rangle)$: the first defines the input parameters of the corresponding function whereas the second the output ones. Each parameter is defined in the traditional way by a pair containing its name and type: (in_i, it_i) for input and (out_i, ot_i) for output parameters. Each interface has a special function name **cons**, standing for the constructor of the smart contract. We use $I[n]$ to denote the signature of function named n.
- The *annotation* $A = (inv, Pre, Post)$ is a triple containing the contract invariant inv, a finite mapping, Pre, from function names to their respective preconditions, and a finite mapping, $Post$, from function names to their corresponding postconditions. The contract invariant is a predicate involving (i.e. parameterised by) the specification variables, whereas the two mappings associate function names to predicates involving the specification member variables as well as the function arguments — Pre predicates involve only the input arguments and the pre-state, whereas $Post$ can involve input and output arguments, and pre-/post-states. These predicates operate on function call arguments (i.e., $e.invals$ and $e.outvals$) and contract states (i.e., Q) as we introduce next. Each function

[2] Typically, specifications might involved user-defined types such as structs and enumerations. For the sake of conciseness and generality, our specification definition omits the explicit declaration of such types and assume that the types used in a specification, even when user-defined, are well known.

defined in I must have a *Pre* and *Post* mapping. We use *Pre*[n] and *Post*[n] to denote the pre- and post-conditions of a function named n.

The semantics of a specification is given by the following labelled transition system (LTS). Roughly speaking, the behaviour of a specification is given by an LTS where the states capture the values of the contract variables and the transitions are labelled by function calls that must obey the classical semantics of the associated contract annotations.

Definition 2 The labelled transition system $L = (\hat{s}, Q, \Sigma, \Delta)$ induced by the specification (D, I, A) is given by:

- $\hat{s}$ is the *distinguished* starting state; it is not a proper state in the sense that it is not an assignment to the values of the smart contract variables — it merely represents an uninitialised/uncreated smart contract.
- Let $D = \langle (n_1, t_1), \ldots, (n_{|D|}, t_{|D|}) \rangle$. The state space is given by $Q = (t_1 \times \ldots \times t_{|D|}) \cup \{\hat{s}\}$. For s state $s \in Q$, we use the notation $s[n_i]$ to denote the value in state s of the variable named n_i;
- Σ is the set of function calls induced by I. A function call $e \in \Sigma$ is a triple $(n, is = \langle iv_1, \ldots, iv_{|is|} \rangle, os = \langle ov_1, \ldots, ov_{|os|} \rangle)$ such that n is the name of a function in I, and input and output values iv_i and ov_i conform to the signature $I[n]$ in the usual way. For a function call e, we use $e.name$ to get the function name, $e.invals$ to get the input values, and $e.outvals$ to get the output values.
- The transition relation is given by $\Delta \subseteq Q \times \Sigma \times Q$ such that:
  - For $s \neq \hat{s}$, $e.name \neq$ **cons**, and $s' \neq \hat{s}$, with $pre = Pre[e.name]$ and $post = Post[e.name]$, there is a transition $(s, e, s') \in \Delta$ iff:

$$(pre(s, e.invals) \land inv(s)) \Rightarrow (post(s, s', e.invals, e.outvals) \land inv(s')).$$

  - For $s = \hat{s}$, $e.name =$ **cons**, and $s' \neq \hat{s}$, with $pre = Pre[e.name]$ and $post = Post[e.name]$, there is a transition $(s, e, s') \in \Delta$ iff:

$$pre(s, e.invals) \Rightarrow (post(s, s', e.invals, e.outvals) \land inv(s')).$$

No other transition is in Δ. Note that *inv* is a predicate on the state Q, a precondition *pre* operates on the function's pre-state ($s \in Q$) and input arguments ($e.invals$), and a postcondition *post* on the function's pre- and post-states ($s, s' \in Q$) and its input and output arguments ($einvals$ and $e.outvals$, respectively).

Note that the starting state is a modelling device that ensures we have a single starting state from where only constructors can be called. Thus, the pre- and postconditions for the constructor function **cons** do not constrain the pre-state (i.e. the starting state) of the call. We leave the pre-states as part of the predicates for the sake of generality and notation consistency so pre/post-conditions and invariants operate on Q.

We also observe that, if a precondition is not met, a transition can lead to a (non-starting) arbitrary state. We use the notation $s \xrightarrow[\Delta]{e} s'$ as a shorthand for $(s, e, s') \in \Delta$; we omit the Δ subscript when the transition relation is clear from the context.

A path for this system is represented by a sequence $\langle s_1, e_1, \ldots, s_k, e_k, s_{k+1} \rangle$ such that $s_i \xrightarrow{e_i} s_{i+1}$ for $i \in \{1, \ldots, k\}$; a path must involve at least one transition. A path starting from the starting state is a *starting path*.

Our notion of specification refinement follows well-accepted principles: it allows for invariant and postcondition strengthening whereas preconditions can be weakened.

Definition 3 A specification $S = (D, I, A)$ is refined by a specification $S' = (D', I', A')$, with induced state spaces Q and Q', respectively, considering an abstraction function α from Q' to Q, denoted by $S \sqsubseteq_\alpha S'$, *iff*:

- $\forall s \in Q' \bullet inv'(s) \Rightarrow inv(\alpha(s))$
- For each function[3] $n \mapsto (ins, outs)$ in I — with $pre = Pre[n]$, $post = Post[n]$, *Ins* and *Outs* the cartesian products of the types for input and output arguments for n:
  - $n \mapsto (ins, outs)$ must be in I'

[3]Recall that the interface I is a mapping from function names to their signatures defined by a pair of sequences: one for the input parameters and the other for the output parameters, with their types. So we use the notation $n \mapsto (in, outs)$ to represent a maplet of the interface I.

- $\forall s \in Q', is \in Ins \bullet pre(\alpha(s), is) \Rightarrow pre'(s, is)$
- $\forall s, s' \in Q', is \in Ins, os \in Outs \bullet$
  $post'(s, s', is, os) \Rightarrow post(\alpha(s), \alpha(s'), is, os)$

This definition provisions for many types of specification evolution. It allows for simple strengthening of postconditions/invariant and weakening of preconditions, whereas it also enables more complex evolutions such as data-representation changes and interface extensions — and even a combination of all of them. The abstraction function allows for changes in the data representation of a contract, whereas our quantification on I (instead of I') in the constraining of annotations allows S' to have extra functions with respect to S. We give examples of all these types of refinement/evolution later.

One question that remains is what guarantee does this notion of refinement provide? We demonstrate three meta-properties conferred by our framework. Properties shown to hold for all states that respect the invariant of a specification are maintained by a refined specification, modulo the abstraction function.

Theorem 1 *Let S and S' be two specifications with induced state spaces Q and Q', and invariants inv and inv', respectively. Additionally, let ϕ be a predicate over states in Q.*

$$\begin{array}{c}\textit{If } \forall s \in Q \bullet inv(s) \Rightarrow \phi(s) \textit{ and } S \sqsubseteq_\alpha S' \textit{ then} \\ \forall s' \in Q' \bullet inv'(s') \Rightarrow \phi(\alpha(s')).\end{array}$$

Proof From $s' \in Q'$ with $inv'(s')$ and $S \sqsubseteq_\alpha S'$, we can derive that $inv(\alpha(s'))$ holds. From $\forall s \in Q \bullet inv(s) \Rightarrow \phi(s)$ and $inv(\alpha(s'))$, we can derive $\phi(\alpha(s'))$. □

Properties shown to hold for a *subset* of states and input values that respect the precondition of a specification's function are maintained by the corresponding function of a refined specification, modulo the abstraction function.

Theorem 2 *Let S and S' be two specifications with induced state spaces Q and Q', respectively. Additionally, let function n be in the interface of both S and S', and pre and pre' be its pre-conditions in specifications S and S', respectively, and Ins the input-arguments space for this function. Finally, let ϕ be a predicate over $Q \times Ins$.*

$$\begin{array}{c}\textit{If } \forall s \in Q, is \in Ins \bullet \phi(s, is) \Rightarrow pre(s, is) \textit{ and } S \sqsubseteq_\alpha S' \\ \textit{then } \forall s' \in Q', is \in Ins \bullet \phi(\alpha(s'), is) \Rightarrow pre'(s', is).\end{array}$$

Proof From $s' \in Q'$ and $is \in Ins$ with $\phi(\alpha(s'), is)$, and $\forall s \in Q \bullet \phi(s) \Rightarrow pre(s)$, we can derive that $pre(\alpha(s'), is)$ holds. From $S \sqsubseteq_\alpha S'$ and $pre(\alpha(s'), is)$, we can derive $pre'(s', is)$. □

Finally, properties shown to hold for all state pairs, input and output values that respect the post-condition of a specification function are maintained by the corresponding function of a refined specification, modulo the abstraction function.

Theorem 3 *Let S and S' be two specifications with induced state spaces Q and Q', respectively. Additionally, let function n be in the interface of both S and S', and post and post' be its post-conditions in specifications S and S', respectively, Ins the input-arguments space for this function, and Outs the output-arguments space. Finally, let ϕ be a predicate over $Q \times Q \times Ins \times Outs$.*

$$\begin{array}{c}\textit{If } \forall s, s' \in Q, is \in Ins, os \in Outs \bullet post(s, s', is, os) \Rightarrow \\ \phi(s, s', is, os) \textit{ and } S \sqsubseteq_\alpha S' \textit{ then} \\ \forall s, s' \in Q' \bullet post'(s, s', is, os) \Rightarrow \phi(\alpha(s), \alpha(s'), is, os)\end{array}$$

Proof From $s, s' \in Q'$, $is \in Ins$, and $os \in Outs$ with $post'(s, s', is, os)$ and $S \sqsubseteq_\alpha S'$, we can infer that $post(\alpha(s), \alpha(s'), is, os)$ holds. From $\forall s, s' \in Q, is \in Ins, os \in Outs \bullet post(s, s', is, os) \Rightarrow \phi(s, s', is, os)$ and $post(\alpha(s), \alpha(s'), is, os)$, we can derive $\phi(\alpha(s), \alpha(s'), is, os)$. □

Thus, by relying on our notion of specification refinement, our framework can ensure the maintenance of these three types of meta-properties.

For the sort of formalism we use in this paper, safety properties are typically derived from invariants that should hold for every reachable state of the system. Note, however, that a specification invariant is not guaranteed to be enforced in every reachable state. If a precondition is not met, for instance, the transition takes the system to an arbitrary state that might not respect the invariant. States that are reachable via function calls where the preconditions are met along the path, however, respect the invariant.

Theorem 4 *If a state s is reachable via a starting path (a path starting at the starting state) $\langle s_1 = \hat{s}, e_1, \ldots, s_k, e_k, s_{k+1} = s\rangle$ of a smart contract whose specification has an established invariant inv, and, for each $i \in \{1, \ldots, k\}$ we have that $pre_i(s, e_i.invals)$ holds, with $pre_i = Pre[e_i.name]$, then $inv(s)$.*

Proof We can prove this by induction on the size of starting paths.

- Base case: For a starting path containing a single transition, we have that the transition has to be a constructor function that established the invariant.
- Inductive step: For a starting path p satisfying that property, we show it holds for the path $p^\frown\langle e, s'\rangle$. Without loss of generality, let us call s the last state in p. We have that $inv(s)$ given that p satisfies the property, and that the precondition is satisfied by e per our assumption. From these two facts and the behaviour of function calls, we can deduce that e maintains the invariant and enforces the postcondition.

□

Typically, verification frameworks employing design-by-contract approaches target modules or classes that are part of a larger and fixed program. In that context, the function calls made to these modules are fixed and their preconditions can be enforced (and are verified to be enforced) in the respective call sites. Hence, the reachable states of an instance of such a module, in this context, satisfy the invariant specified. In the case of smart contracts, their context is usually not fixed. Therefore, this type of verification on call sites to enforce preconditions is not possible. So, although our framework allows arbitrary preconditions, for the kind of open systems verification we explore in our case studies here, we require that preconditions be trivially satisfied, i.e. they must be the constant *true* predicate.

Requirement 1 All preconditions on smart contract specifications must be the constant predicate *true*.

This requirement combined with Theorem 4 ensures that the reachable states of the smart contract satisfy the invariant.

The burden of enforcing the postcondition without resorting to a stronger precondition goes to the implementer of smart contracts, but the Ethereum platform offers constructs that help in this regard. For instance, the Ethereum platform offers commands that revert a transaction if some expected property of a pre-state is not met, acting as a guard.

Language constructs from Solidity and from the *solc-verify* specification language, which in turn borrows elements from the Boogie language [15, 40], are used to create our specifications in practice. 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. 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)`, for Solidity type x, that can only be used in a postcondition, and it denotes the value of `v` in the function's execution pre-state. In this specification, the invariant states that the account for the zero address must have a zero balance associated to it. The postconditions for `deposit` and `withdraw` ensure that the contract balance and account balances are appropriately set.The postconditions for the `constructor` function simply ensures the initial account balances are zero.

Figure 5 illustrates the `ToyWalletNew` specification that refines the specification in Figure 4. The refined specification has a different data representation for its state — an account has associated to it a Boolean value denoting whether it has been opened or not — and an extra function `is_open` that allows accounts to be open. Therefore, this refinement, albeit simple, illustrates both data representation change and interface extension.

We use *solc-verify* to validate the semantic requirements imposed by our refinement relation. We encode the proof obligations associated with invariant and postcondition strengthening and use *solc-verify* to discharge them. For this task, we do not use *solc-verify* in a conventional way, as it provides no direct support for checking specification refinement. Instead, given two smart contracts being tested for refinement, we produce a

```
/** @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.

```
/** @notice invariant accs[address(this)].bal == 0 */
contract ToyWalletNew {

  struct Account {
    uint bal;
    bool is_open;
  }

  mapping (address => Account) accs;

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

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

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

  /** @notice postcondition accs[msg.sender].is_open */
  function open() public;
}
```

Fig. 5: ToyWallet refined specification.

third contract — which we call the *refinement-validity contract* — that encodes the necessary implications between the invariants and postconditions of these contracts so that *solc-verify* can check their validity. In the following, we explain how *refinement-validity contracts* are created and use `ToyWalletRefinement`, presented in Figures 6 and 7, — which is the contract (and encoding) created to validate that `ToyWallet` (Figure 4) is refined by `ToyWalletNew` (in Figure 5) — to illustrate our strategy to check for specification refinement using *solc-verify*.

For specifications S and S', we create the refinement-validity contract as follows. Firstly,

```
contract ToyWalletRefinement {

  struct StateAbs {
    mapping (address => uint) accs;
    address _this;
  }

  struct Account {
    uint bal;
    bool is_open;
  }

  struct StateCon {
    mapping (address => Account) accs;
    address _this;
  }

  address msg_sender;
  address msg_sender_old;

  StateAbs abs;
  StateAbs abs_old;
  StateCon con;
  StateCon con_old;

        // ... snip
}
```

Fig. 6: ToyWalletRefinement contract variables.

the state of this contract must capture both the state of S and S' as our proof obligations define how they are related. So, we create two member elements `abs` and `con` of type `StateAbs` and `StateCon` to captures the states of S and S', respectively. The types are Solidity structures capturing the corresponding data representation of specifications. We also need two additional elements, `abs_old` and `con_old`, to capture pre-states as postconditions might also constrain them. The state must also include other variables that postconditions refer to. Figure 6 depicts the state representation for the refinement-validity contract considering `ToyWallet` and `ToyWalletNew`. Note how the variable (implicit parameter) `msg.sender` is captured by `msg_sender`; its pre-state value being captured by variable `msg_sender_old` in our encoding.

Each (semantic) proof obligation gives rise to a special function in the refinement-validity contract, and the proof obligation itself is encoded as a pre- and postcondition pair on this function. The invariant-strengthening implication $inv' \Rightarrow inv$ (proof obligation) gives rise to the function `inv`. The precondition of this function is $inv' \wedge \alpha$ whereas the postcondition is inv; here, we use α as the characteristic predicate of the abstraction function. Predicates inv and inv' are straightforwardly translated to constrain the refinement-validity state element corresponding to the contract's original state, whereas α is expected to be given in terms of the refinement-validity state already. For instance, `ToyWalletNew`'s invariant `accs[address(this)].bal == 0` is encoded as precondition `con.accs[address(con._this)].bal == 0` of function `inv` in Figure 7; `accs` and `this` are encoded as `con.accs` and `con._this`, respectively. For this refinement, the predicate α is given by the preconditions annotated with "// α" in function `deposit_post` of `ToyWalletRefinement`; for the sake of clarity, we omit some of the conjuncts of this predicate when they are obviously irrelevant.

The postcondition-strengthening implication $post' \Rightarrow post$ for function f gives rise to the function f`_post` with precondition $post'$ (apart from the encoding of the abstraction function) and postcondition $post$ accounting for a translation that is similar to that of invariants. Postcondition references to the pre-state (i.e. expressions of the form `__verifier_old_`x`(v)`), which cannot appear on invariants, are translated to constrain the corresponding pre-state element. For instance, postcondition `address(this).balance == __verifier_old_uint(address(this).balance)+ msg.value` of function `deposit` in `ToyWalletNew` is translated into the postcondition `address(abs._this).balance == address(abs_old._this).balance + msg_value` of function `deposit_post` in `ToyWalletRefinement`. For preconditions, as we require preconditions to be the predicate *true*, the encoding of this proof obligation is trivial. However, it is worth mentioning that the reverse implication is captured analogously, and we add these trivially-satisfied obligations to our `ToyWalletRefinement` contract for the sake of completeness.

The verifier function *verify-creation* (see Figure 3) does not carry out any sort of specification refinement; only the initial *reference specification* is set at that time. The *verify-upgrade* function checks whether the specification passed as an argument is a refinement of the current specification, taking into account the abstraction function passed as an argument. It carries out basic structural validations to check the syntactic obligation imposed by our refinement notion

```
contract ToyWalletRefinement {

  // ... snip

  /** @notice precondition con.accs[address(con._this)].bal == 0
  * @notice precondition forall (address addr) con.accs[addr].bal == abs.accs[addr] // α
  * @notice precondition con._this == abs._this  // α
  * @notice postcondition abs.accs[address(abs._this)] == 0 */
  function inv() public {}

  /** @notice precondition true
  * @notice postcondition true */
  function deposit_pre() public {}

  /** @notice precondition address(con._this).balance == address(con_old._this).balance + msg_value
  * @notice precondition con.accs[msg_sender].bal == con_old.accs[msg_sender_old].bal + msg_value
  * @notice precondition forall (address addr) addr == msg_sender || con_old.accs[addr].bal == con.
      accs[addr].bal
  * @notice precondition forall (address addr) con.accs[addr].bal == abs.accs[addr] // α
  * @notice precondition con._this == abs._this // α
  * @notice precondition forall (address addr) (con_old.accs[addr].bal == abs_old.accs[addr]) // α
  * @notice precondition con_old._this == abs_old._this // α
  * @notice postcondition address(abs._this).balance == address(abs_old._this).balance + msg_value
  * @notice postcondition abs.accs[msg_sender] == abs_old.accs[msg_sender_old] + msg_value
  * @notice postcondition forall (address addr) addr == msg_sender || abs_old.accs[addr] == abs.accs[
      addr] */
  function deposit_post(uint msg_value) public {}

  //... withdraw_pre and withdraw_post ommitted for conciseness
}
```

Fig. 7: ToyWalletRefinement contract functions.

and the refinement-validity-contract construction to discharge semantic obligations.

3.1.2 Implementation conformance

In the previous section, we have already defined a smart contract specification. Before we define a relation that captures whether an implementation meets (conforms to) a specification, we need to define an abstract notion of a smart contract implementation, which, from now on, we will also refer to simply as a smart contract.

Definition 4 A smart contract is given by a pair $C = (D, F)$ where D describes its member variables — in the same way we do for specifications — and F consists of member function declarations. As for I in the specification, F is a mapping from function names into a signature $(ins, outs)$ and its body bd. We assume the existence of function $semantics(C)$ that constructs LTS $L = (\hat{s}, Q, \Delta, \Sigma)$ based on the contract definition C where $\hat{s}$, Q and Σ are obtained as per Definition 2 — Σ takes into account signatures in F rather than I — and $\Delta \subseteq Q \times \Sigma \times Q$.

The conformance relation is then given as follows.

Definition 5 A smart contract implementation $C = (D', F')$, with $(\hat{s}, Q', \Delta', \Sigma') = semantics(C)$, conforms to specification $S = (D, I, A)$ with induced LTS $(\hat{s}, Q, \Delta, \Sigma)$ *iff* $D = D'$, I and F declare the same functions with matching signatures, and $\Delta \supseteq \Delta'$. This relation is denoted by $S \leq C$.

Similarly to specification refinement, conformance ensures that an implementation "inherits" some properties of the specification. So, a property that is derived from a specification invariant is inherited by the specification. This corollary and the next one follow from Definition 5.

Corollary 5 *Let S and C be a specification and an implementation with induced state spaces Q and Q', respectively, and where inv is the invariant of S. Additionally, let ϕ be a predicate over states in Q.*

$$\text{If } \forall s \in Q \bullet inv(s) \Rightarrow \phi(s) \text{ and } S \leq C \text{ then } \forall s' \in Q' \bullet inv(s') \Rightarrow \phi(s').$$

Moreover, a property that is true for transitions satisfying a function postcondition is also inherited.

Corollary 6 *Let S and C be a specification and an implementation with (induced) transition spaces Δ and Δ', respectively. Additionally, for any function f in the interface of S with post its post-conditions in specification S, Ins the input-arguments space for this function, and Outs the output-arguments space. Finally, let ϕ be a predicate over $Q \times Q \times Ins \times Outs$.*

$$\begin{gathered} \text{If } \forall (s, e, s') \in \Delta \bullet e.name = \\ f \wedge post(s, s', e.invals, e.outvals) \Rightarrow \\ \phi(s, s', e.invals, e.outvals) \text{ and } S \leq C \text{ then} \\ \forall (s, e, s') \in \Delta' \bullet e.name = \\ f \wedge post(s, s', e.invals, e.outvals) \Rightarrow \\ \phi(s, s', e.invals, e.outvals) \end{gathered}$$

In practice, we consider smart contracts written in Solidity and targeting the Ethereum platform. As 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, we rely on other works that propose formalisations for Solidity and Ethereum [32, 11, 73]. In particular, our focus is on using the modular verifier *solc-verify* to discharge the semantic obligations imposed by our conformance notion. In our framework, an implementation can be annotated with loop invariants that help the prover discharge obligations.

The *verify-creation* function (see Figure 3) works as follows. Firstly, the syntactic obligation imposed by Definition 5 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. Figure 8 illustrates a merged contract that takes into account the specification in Figure 4; for conciseness purposes, we do not present the implementation in a separate figure as it is cleanly embedded into the merged contract itself. 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* succeeds. Function *verify-upgrade* is implemented in a very similar way as far as implementation-satisfiability checking.[4]

3.1.3 State transformation soundness

Unlike traditional class refinement or behavioural subtyping frameworks that check whether *types* preserve some properties as they possibly evolve into alternative representations, our framework deals also with persistent object-s/values of such types. Smart contract instances are persistent objects and our framework must also ensure that the current state of the smart contract is updated into its new representation; traditional frameworks do not take persistence into account.

Our framework requires, for each update, an algorithmic concretisation function, which we call `init`, to transform abstract states into concrete ones. As a convention, we use Solidity to implement an `Init` contract with a single `init` function that carries out this transformation. The proof obligation that we impose on this function is that it must be an inverse of the abstraction function; by this obligation we automatically require that the abstraction function be bijective. We plan to lift this requirement in future incarnations of our framework which would use an abstraction relation instead. For instance, in Figure 9, we have an `init` function that converts an abstract state captured by variable `storage_abs` into a concrete state captured by `storage_con`; this template with a type `FacetStorageAbs` (representing the structure of the abstract state) and `FacetStorageCon` (representing the structure of the concrete state) can be generically applied to carry out such translations. The postcondition captures the desired proof obligation, namely that if one translates the value of `storage_con`, after this function has executed, it must give back the value of `storage_abs` in the pre-state. This example also illustrates how loops can be annotated with invariants to help the prover.

Our framework uses Solidity and *solc-verify* to implement this *state-transformation* component of our framework. However, it should be pointed out that Solidity and *solc-verify*, in their current form, might not be the best alternative for this purpose in a fully-fledged framework. There are some features of the language and prover

[4] The function *verify-upgrade* can be optimised as follows. Unlike *verify-creation*, this function can assume that the constructor's obligations have been met. The constructor is only executed at contract-creation time. The upgrade process need only to check for conformance for the implementation of the (non-constructor) public functions.

```
 /** @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 {
    require(msg.sender != address(this));
    accs[msg.sender] = accs[msg.sender] + msg.value;
  }

  /** @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);
    require(msg.sender != address(this));
    bool ok;
    ok = msg.sender.send(value);
    if (!ok){
      revert();
    }
    accs[msg.sender] = accs[msg.sender] - value;
  }
}
```

Fig. 8: ToyWallet merged contract.

that make writing such translations unnecessarily complicated. For instance, the keys that have been used in a mapping in Solidity cannot be obtained by some built-in language construct, like an iterator over the pairs effectively stored in the mapping. So, one has to implement such functionality for mappings, which is essential, for instance, to convert a mapping into some other type of representation, like a pair of arrays. Otherwise, one is likely to have to iterate over an entire set of mapping keys that could reach 2^{256} values, since there is no predefined way to identify the pairs stored in a mapping in Solidity.

Given an `Init` contract and an abstraction function, α, passed as arguments, the function *verify-upgrade* (see Figure 3) annotates the `init` function with the appropriate proof obligation, as explained, and uses *solc-verify* to discharge it.

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* [65, 52], 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 the diamond pattern [52] to deploy *upgradable* contracts; this pattern calls the implementation instance a *facet*. While most of the contracts that we use for verification are of Solidity version 0.5.∗, the contracts in this section are of version 0.6.∗. The diamond proxy relies on some Solidity primitives that are only available from this version onward.

The upgrader relies on the Solidity contract `Proxy` as per Figure 10. The constructor of the proxy sets the initial implementation with the `_facet` argument and calls the constructor `cons`

```
contract Init {

    struct AddrBal {
        address addr;
        uint bal;
    }

    struct FacetStorageAbs {
        AddrBal[] accs;
    }

    struct AddrBalIsOpen {
        AddrBal addr_bal;
        bool is_open;
    }

    struct FacetStorageCon {
        AddrBalIsOpen[] accs;
    }

    FacetStorageAbs storage_abs;
    FacetStorageCon storage_con;

    /** @notice postcondition forall (uint i) i < 0 || i >= storage_abs.accs.length || storage_con.accs
        [i].addr_bal.addr == storage_abs.accs[i].addr
    * @notice postcondition forall (uint i) i < 0 || i >= storage_abs.accs.length || storage_con.accs[i
        ].addr_bal.bal == storage_abs.accs[i].bal
    * @notice modifies storage_con */
    function init() public {
        storage_con.accs.length = storage_abs.accs.length;

         /** @notice invariant forall (uint j) j < 0 || j >= i || storage_con.accs[j].addr_bal.addr ==
             storage_abs.accs[j].addr
          * @notice invariant forall (uint j) j < 0 || j >= i || storage_con.accs[j].addr_bal.bal ==
             storage_abs.accs[j].bal */
        for (uint i = 0; i < storage_abs.accs.length; i++) {
            storage_con.accs[i].addr_bal.addr = storage_abs.accs[i].addr;
            storage_con.accs[i].addr_bal.bal = storage_abs.accs[i].bal;
            storage_con.accs[i].is_open = true;
        }
    }

}
```

Fig. 9: Example of init function.

function[5] on the constructor contract at `_cons` to initialise the storage of that contract. The update function allows the `facet` address to be updated and an `init` function at address `_init` can be called to transform the proxy's storage. Note that both `constructor` and `update` can only be called by the trusted deployer; this is enforced by the statement `require(msg.sender ==` $addr_{td}$`)`, where $addr_{td}$ is the well-known address of the trusted deployer. In the process of creating and upgrading contracts, the trusted deployer acts as an external participant of the Ethereum platform identified by $addr_{td}$. Finally, the `fallback` function simply delegates to the facet the calls to the proxy. This delegation is carried out by the `delegatecall` low-level command that dynamically borrows and executes the code at `facet` but it reads and/or modifies the state (i.e. storage) of the `proxy` instance. 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 fallback function is executed whenever the signature of the call does not match any of the other public contract functions. So, any call that does not target `upgrade` is delegated to the facet. This means that if the facet has a function with the same signature as `update` — function-call dispatching is made based on function signatures — it can never

[5]We are assuming that the constructor function has no parameters for the sake of simplicity. Our framework could easily be adapted to include such parameters, but also, given that this function is only called once, its expected arguments could be hardcoded into the constructor code turning it into a function without parameters.

```
contract Proxy {

address facet;

constructor(address _facet, address _cons) public {
    require(msg.sender == addr_td);
    facet = _facet;
    (bool ok,) = _cons.delegatecall(abi.encodeWithSignature("cons()"));
    if (!ok) revert();
}

function update(address _facet, address _init) public {
    require(msg.sender == addr_td);
    facet = _facet;
    if (_init != address(0)){
        (bool ok,) = _init.delegatecall(abi.encodeWithSignature("init()"));
        if (!ok) revert();
    }
}

fallback() external payable {
  address _facet = facet;

    assembly {
        calldatacopy(0, 0, calldatasize())
        let result := delegatecall(gas(), _facet, 0, calldatasize(), 0, 0)
        returndatacopy(0, 0, returndatasize())
        switch result
          case 0 {revert(0, returndatasize())}
          default {return (0, returndatasize())}
    }
}
}
```

Fig. 10: Proxy contract.

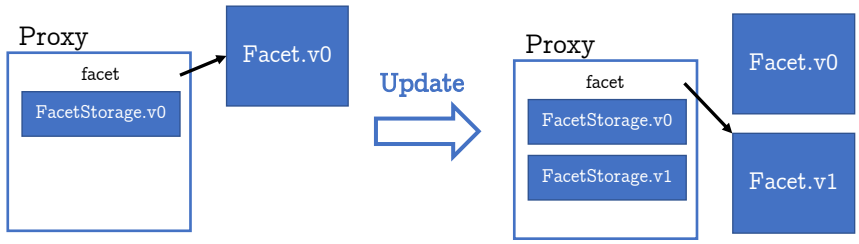

Fig. 11: Simplified proxy upgrade process.

be called through the proxy. Figure 11 depicts a simplified and intuitive version of the update process, which focus on how the proxy's `facet` attribute and storage, and facet contracts are upgraded.

A facet captures an implementation instance. For a contract C, we use *facet*(C) to denote the contract generated from it. In Figure 12, the `Facet` contract captures the implementation for our `ToyWallet` presented in Figure 8; i.e. *facet*(`ToyWallet`). We use the *diamond storage* approach to capture the member variables of a contract so that it is possible to dynamically change the data representation used by our upgradable contract. This approach uses a storage offset represented by constant `FACET_STORAGE_POSITION` and calculated based on the string `"toywallet.storage.v0"` to designate where the state (i.e. member variables) of the current facet, captured by `struct FacetStorage`, is stored. By changing this string, say to `"toywallet.storage.v1"`, the offset changes and we can "allocate" another storage block with a possibly new data representation for the contract. The function `getStorage()` yields the pointer to this storage offset. Note that the code of `deposit` and `withdraw` — and more generally of C functions being implemented in a facet — is modified

```
contract Facet {

bytes32 constant FACET_STORAGE_POSITION = keccak256("toywallet.storage.v0");

struct FacetStorage {
     mapping (address => uint) accs;
}

function getStorage() internal pure returns (FacetStorage storage storageStruct) {
    bytes32 position = FACET_STORAGE_POSITION;
    assembly {
        storageStruct_slot := position
    }
}

function deposit (uint _garb) payable public {
    require(msg.sender != address(this));
    getStorage().accs[msg.sender] = getStorage().accs[msg.sender] + msg.value;
}

function withdraw (uint value) public {
    require(getStorage().accs[msg.sender] >= value);
    require(msg.sender != address(this));
    bool ok;
    (ok,) = msg.sender.call("");
    if (!ok){
        revert();
    }
    getStorage().accs[msg.sender] = getStorage().accs[msg.sender] - value;
}
}
```

Fig. 12: Facet contract.

```
contract Constor {

    bytes32 constant FACET_STORAGE_POSITION = keccak256("toywallet.storage.v0");

    struct FacetStorage {
         mapping (address => uint) accs;
    }

    function getStorage() internal pure returns (FacetStorage storage storageStruct) {
        bytes32 position = FACET_STORAGE_POSITION;
        assembly {
            storageStruct_slot := position
        }
    }

    function cons() payable public {
        // some code to initialise getStorage() pointer
    }

}
```

Fig. 13: Constructor contract.

to use `getStorage()` as a way to access and modify the state of the facet. Note as well that a facet can be replaced by another one that has additional functions — the pattern would allow even the removal of functions but our framework forbids that. The fallback-function-based dynamic dispatching used by this pattern allows for this sort of behavioural extension as it simply forwards (non-update) calls to the facet, which, in turn, processes the call by executing the appropriate function's code, if it exists.

Figure 13 illustrates the constructor `Constor` contract for `ToyWallet`. For contract C, we use $constructor(C)$ to denote the constructor contract generated for C. It relies on the same diamond storage approach to access and modify the storage

of the proxy contract. The `ToyWallet` constructor is empty and, because of that, so is the `cons` function. Nevertheless, we do point out where the constructor code would be inserted.

Figure 14 illustrates an `Init` contract. For `Init` contract C, we denote by $init(C)$ the contract modified to use the diamond storage approach. It was generated based on the one in Figure 9, using the diamond storage approach. Note how the `init` function works with two "versions" of the storage and performs a conversion between them.

The upgrader function *create-contract*(C) behaves as follows. Firstly, it issues transactions *create-contract*($facet(C)$) and *create-contract*($constructor(C)$) to the Ethereum platform to create the initial implementation instance and the constructor contract at addresses $\texttt{addr}_{impl}$ and $\texttt{addr}_{cons}$, respectively. Secondly, it issues transaction *create-contract*(`Proxy`, *args*), such that `_facet` is set to $\texttt{addr}_{impl}$ and `_cons` is set to $\texttt{addr}_{cons}$, to create the proxy instance at address $\texttt{addr}_{pro}$. 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 first step issues a *create-contract*(`Init`) to deploy this contract at address $\texttt{addr}_{ini}$ instead of deploying the constructor contract, and the second step issues transaction *call-contract*($\texttt{addr}_{pro}$,`upgrade`, *args*), triggering the execution of function `upgrade` in the proxy instance and changing its `_facet` address to the new implementation instance and passing $\texttt{addr}_{ini}$ via `_init`.

3.3 Trusted registry

An external participant in the Ethereum platform can confirm that a given instance was created by the trusted deployer by calling its *get-spec* function. Smart contracts in the Ethereum platform, however, cannot probe external services, such as the trusted deployer. Therefore, an Ethereum smart contract would have no means to check whether an instance that it wants to interact with is safe. As *composability*, namely, this ability of smart contracts to interact and cooperate, is one of the main features of Ethereum, we propose a mechanism by which contracts can programmatically test if a counterpart contract is safe. As part of the trusted deployer infrastructure, we create a *trusted registry* as an instance of the `TrustedRegistry` contract, given in Figure 15, deployed at the trusted address $\texttt{addr}_{reg}$. Its variable `verified_addr` mirrors the trusted deployer's internal registry. Only the trusted deployer — note the condition `msg.sender ==` $addr_{td}$ — can call `add_mapping` to update the registry but any contract can call `get_spec` to access the registry. 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. This trusted registry has other benefits in its own right. For instance, given its implementation as an Ethereum smart contract, it inherits the non-functional properties offered by this platform such as high-availability and secure execution.

Having this trusted registry might seem a small contribution but it brings significant guarantees that cannot be otherwise obtained. In Solidity, and generally in Ethereum, *type* information about contract instances cannot be obtained programmatically in a trusted way. For instance, when executing a function call, a smart contract cannot check whether the target contract has code associated with the specific function it is trying to call. It simply tries to execute the code associated with a function signature. Solidity's behaviour can be especially problematic in this aspect. They have an implicit execution flow by which a special *fallback* function is executed if none of the other functions in the target contract have the intended signature. Thus, a smart contract might execute a function on a counterpart that apparently successfully terminates when in fact a completely different unwanted function runs instead. Our trusted deployer acts as a *verification oracle* that pushes into the blockchain, and specifically into the `TrustedRegistry` contract, information not only about the interface of deployed contracts, ensured by the syntactic obligation in Definition 5, but also about their behaviour, ensured by the semantic obligation on the same definition. This information can be programmatically obtained via the `get_spec` function in `TrustedRegistry`, of course. Amongst other usages, this ability can be harnessed to create a sort of safe contract call. A contract can check that the to-be-called counterpart meets a certain specification and only issues the call if so. For instance, let us assume that we

```
contract Init {

    struct AddrBal {
        address addr;
    uint bal;
  }

    struct AbsFacetStorage {
        AddrBal[] accs;
    }

    struct AddrBalIsOpen {
        AddrBal addr_bal;
    bool is_open;
  }

    struct ConFacetStorage {
      AddrBalIsOpen[] accs;
    }

    bytes32 constant ABS_FACET_STORAGE_POSITION = keccak256("toywallet2.storage.v0");

    bytes32 constant CON_FACET_STORAGE_POSITION = keccak256("toywallet2.storage.v1");

    function getAbsStorage() internal pure returns (AbsFacetStorage storage storageStruct) {
        bytes32 position = ABS_FACET_STORAGE_POSITION;
        assembly {
            storageStruct_slot := position
        }
    }

    function getConStorage() internal pure returns (ConFacetStorage storage storageStruct) {
        bytes32 position = CON_FACET_STORAGE_POSITION;
        assembly {
            storageStruct_slot := position
        }
    }

    function init() public {
        for (uint i = 0; i < getAbsStorage().accs.length; i++) {
            getConStorage().accs.push();
        }

        for (uint i = 0; i < getAbsStorage().accs.length; i++) {
            getConStorage().accs[i].addr_bal.addr = getAbsStorage().accs[i].addr;
            getConStorage().accs[i].addr_bal.bal = getAbsStorage().accs[i].bal;
            getConStorage().accs[i].is_open = true;
        }
    }
}
```

Fig. 14: Init contract.

want to write a smart contract function that calls `do_something` in contract c, but only does so if c meets specification *expected_spec*. This behaviour could be achieved by the following snippet:

```
requires(TrustedRegistry(addr_reg).get_spec(
    address(c)) == expected_spec);
c.do_something(a, b);
```

A similar approach can be taken by external applications that want to use a safe smart contract. They can instead use the function *get-spec* of the trusted deployer, and abort their execution if the contract they want to interact with does not meet the expected specification.

3.4 Framework guarantees and limitations

Given the validation requirements we impose, our framework offers the guarantee that any upgrade must respect, by transitivity, the *reference specification*, i.e. the specification chosen at deployment time. Upgrades, both of specifications and implementations, must abide by the invariants set in the reference specification. This sort of guarantee means that contract stakeholders can be certain

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

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

Fig. 15: Trusted registry.

that properties derived from the reference specification invariants are maintained by any upgrade. It follows from Theorem 1 and Corollary 5.

Corollary 7 *Let $S_0, \ldots, S_n$ be a series of specifications, $\alpha_0, \ldots, \alpha_{n-1}$ a series of abstraction functions, and C an implementation. Moreover, let Q_0 and Q be the state space of S_0 and C, respectively, and inv the invariant of S_0. Additionally, ϕ is a predicate over states in Q_0, and $\bar{\alpha} = \alpha_1 \circ \ldots \circ \alpha_{n-1}$*

$$\begin{gathered} \textit{If } \forall s \in Q_0 \bullet inv(s) \Rightarrow \phi(s), \\ \forall i \in \{0, \ldots, n-1\} \bullet S_i \sqsubseteq_{\alpha_i} S_{i+1}, \textit{ and } S_n \leq C \textit{ then} \\ \forall s' \in Q \bullet inv(\bar{\alpha}(s')) \Rightarrow \phi(\bar{\alpha}(s')). \end{gathered}$$

Note that the abstraction function is being applied to the states of an implementation. This application is allowed given that the an implementations has the same state space as a specification it conforms to.

Unsurprisingly, our framework also ensures that properties derived from a reference specification postcondition are maintained by upgrades. It follows from Theorem 3 and Corollary 6.

Corollary 8 *Let $S_0, \ldots, S_n$ be a series of specifications, $\alpha_0, \ldots, \alpha_{n-1}$ a series of abstraction functions, and C an implementation. Moreover, let Δ_0 and Δ be the transition spaces for S_0 and C, respectively. For any function f in the interface of S_0 with post its postconditions in specification S_0, Ins the input-arguments space for this function, and Outs the output-arguments space, and let ϕ be a predicate over $Q_0 \times Q_0 \times Ins \times Outs$.*

$$\begin{gathered} \textit{If } \forall (s, e, s') \in \Delta \bullet e.n = \\ f \wedge post(s, s', e.invals, e.outvals) \Rightarrow \\ \phi(s, s', e.invals, e.outvals), \\ \forall i \in \{0, \ldots, n-1\} \bullet S_i \sqsubseteq_{\alpha_i} S_{i+1}, \textit{ and } S_n \leq C \textit{ then} \\ \forall (s, e, s') \in \Delta' \bullet e.name = \\ fn \wedge post(\bar{\alpha}(s), \bar{\alpha}(s'), e.invals, e.outvals) \Rightarrow \\ \phi(\bar{\alpha}(s), \bar{\alpha}(s'), e.invals, e.outvals) \end{gathered}$$

The focus of this paper is examining the underlying verification and upgrade mechanisms used by the trusted deployer rather than presenting the protocol and mechanisms used to secure the trusted deployer service itself. Nevertheless, for the sake of completeness, we briefly discuss some of these aspects here. In this paper, we assume that the trusted deployer is implemented by a trusted third party. So, we assume it correctly executes our framework, making the appropriate verification, deployment and update steps. Moreover, we assume that the blockchain, in our case Ethereum, is highly available and processes the transactions issued by the trusted deployer in a timely manner. Furthermore, for each contract deployed, the trusted deployer identifies a maintainer which has the role of deploying and upgrading the contract's specification and implementation whenever is needed and only this maintainer can evolve this contract. Of course, these evolutions have to respect our verification framework. Thus, our framework is intended to limit the ability of the maintainer in arbitrarily update the code of a smart contract. It can only do so within the refinement/conformance restrictions imposed by our framework. In a subsequent paper, we plan to detail an implementation of the trusted deployer using TEEs and cover the security properties of this service itself. More specifically, we are currently looking at an implementation of the trusted deployer using AMD SEV [61] with an adaptation of the protocol proposed in [9]. In this paper, we also plan to discuss consistency properties related to the trusted deployer state and the blockchain state of its managed contracts.

Connecting the informal trust model that we just presented and our corollaries, we have that our framework proposes a paradigm shift in which contract stakeholders would put their trust on function postconditions and, more importantly, on contract invariants. That is, they would decide to trust a contract based on the properties that can be derived from these specification elements, as any subsequent upgrade would still enjoy these properties. Even a malicious maintainer could not

mislead the trusted deployer into carrying out an unsafe upgrade, violating the expected properties. Of course, if postconditions and invariants are too loose and do not imply the desired properties, the maintainer has room to change the code in potentially harmful ways.

We consider the validation of specifications a problem that is orthogonal to the safe evolution of smart contracts. Moreover, as they are more abstract and typically declarative, specifications can more directly capture the intent of developers — and so they are more likely to reflect the true intent of the developer — as opposed to code. Thus, we use specifications as the main artifacts upon which our framework is based and rely on existing work to cope with specification validation [37, 44, 64, 24]. Once a specification is validated, its evolutions, through formal refinement, preserve the desired properties.

Our framework was inspired by and targets the Ethereum platform. However, the ideas it implements are more general. In particular, the principle of having formal verification enforcing the safe evolution of smart contract instances could be applied to other smart contract platforms.

As smart contracts are long-term executable artifacts, they might reach a point where the evolution cannot be captured by a refinement. For instance, drastic and unplanned changes on the requirements of the contract. In such case, there are *governance* mechanisms that could be put in place to bypass the refinement controls and allow for non-refinement contract evolutions, namely, a committee of stakeholders can implement a voting procedure to break away from the safe-evolution paradigm. Addressing such cases, however, is out of the scope of this paper.

We should also point out that our framework collapses into the one presented in [10] when specifications are unchanged, i.e. upgrades involve only a change in implementation. Verifying specification refinement and carrying out a data transformation are not necessary in this case and the framework can be optimised to work like that one.

One limitation of our current approach is its use of partial correctness notions; we do not yet attempt to show, for instance, that functions terminate. We will extend the framework to support *total correctness* reasoning in the future. Thus, currently, our framework demonstrates that contracts are *safe* but not necessarily *live*. Therefore, in its current form the framework would allow a malicious maintainer to attack a contract, for example, by making all its functions always revert. This attack would effectively lock the funds managed by the contract. We intended to extend our framework to guarantee liveness in the future by looking, especially, at function termination. Nevertheless, the class of safety properties is broad as they, intuitively, capture that "no bad behaviour can arise" as opposed to "some good behaviour will eventually happen" for liveness.

Many frameworks for the analysis of smart contracts focus on looking for specific behavioural patterns underpinning security vulnerabilities [45, 51, 43]. Our work, however, focus on the verification of general safety properties. The advantage of our approach is that it can identify both known and, more importantly, *unknown* behavioural patterns that can lead to a safety violation. For instance, the implementation of our approach, which we detail next, can identify that a reentrancy bug is causing a postcondition or invariant to be violated. Moreover, there may be behavioral patterns that although look problematic might not affect the safety of a contract. For example, there may be reentracy bugs that may not be exploitable and, thus, are harmless.

4 Case Studies

The presentation of our case studies is structured into the following sections. In Section 4.1, we consider the simpler scenario of smart contract evolution that must obey a fixed specification. More elaborate scenarios involving data refinement and interface extension are addressed in Section 4.2. We conclude with Section 4.3 on the limitations and threats to validity.

All the instructions and artifacts (including specifications, sample contracts, and scripts) used in these case studies can be found in the paper's repository.[6]

4.1 Implementation Evolution of a Fixed Specification

For this evolution scenario, we consider the following Ethereum smart contract standards: ERC20, ERC3156 and ERC1155. First, we describe the

[6] https://github.com/formalblocks/safeevolutionrefinement

context (Section 4.1.1) of the evaluation, next we provide the results (Section 4.1.2), and, finally, we detail the analysis of a scenario based on the 0xMonorepo repository; this is used to illustrate a tool — available in the paper's repository — we developed to mechanise the steps of our trusted deployer framework in the case of implementation evolutions with a fixed specification (Section 4.1.3).

4.1.1 Context

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 trustworthy 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: *Does the proposed framework improve on the process of creation and evolution of 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 a 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 three Ethereum smart contract standards: ERC20, ERC3156 and ERC1155. Each of these standards defines a contract interface and is accompanied by an informal description of its functions' behaviours. We chose these standards because they are widely used and in an advanced state of maturity. From 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 to verify the feasibility of the framework, and to evaluate its effectiveness — which can be measured by the number of errors found or safe evolutions that have been proven correct — and its 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

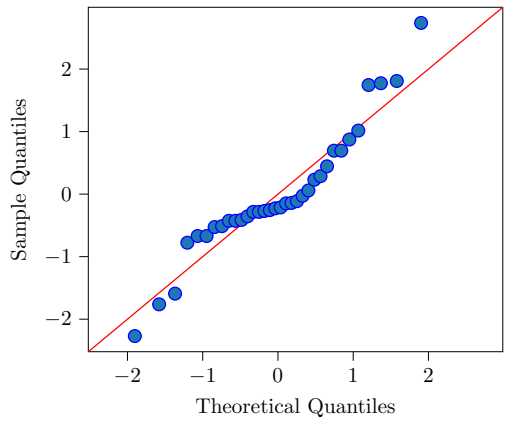

Fig. 16: Q-Q plot — distribution of the analysis execution time.

generated by the Solidity compiler [4] 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, ERC3156 and the ERC1155 token standards. The contract samples we analysed were extracted from 12 github repositories that were public, and presented reasonably complex commit histories 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.

4.1.2 Results

Our framework was able to identify errors of the following categories: Integer Overflow and Underflow (IOU); Non Specification Conformance (NSC), when a function does not meet a specific mandatory requirement defined in its ERC specification; Nonstandard Token Interface (NTI), when the contract does not meet the syntactic

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
ERC3155							
ArgoBytes	c3c92c	4.59s	No errors	ERC3156	512268	3.79s	NSC
ArgoBytes	e524c1	3.19s	No errors	Wrappes	70b977	6.65s	No errors
ArgoBytes	738d47	4.35s	No errors	Wrappes	5f65ac	3.23s	No errors
ArgoBytes	3409ee	2.57s	No errors	Wrappes	6bf6d6	4.79s	No errors
Dss Flash	f2ca83	5.15s	NSC	Weth10	34c42c	5.14s	NSC
Dss Flash	b1e01d	3.37s	NSC	Weth10	b85345	3.85s	NSC
Dss Flash	2e70bb	6.12s	NSC	Weth10	dbe412	3.97s	No errors
Dss Flash	18caa8	4.24s	No errors	Weth10	d2c480	4.25s	No errors
ERC3156	e18bdf	4.01s	NSC	Weth10	4fcc9e	5.39s	No errors

Table 2: ERC20, ERC3156 and ERC1155 Results

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 2 shows the complete list of all results we obtained.

In our study, we encountered the challenge of working with a relatively small sample size for benchmarking various time durations. However, before proceeding with comparisons to existing literature, we took a careful approach to assess our data distribution. To do this, we generated a Quantile-Quantile (Q-Q) plot in Figure 16, a graphical tool used to visualize how well a dataset aligns with a theoretical normal distribution. Upon inspecting the Q-Q plot, we observed that our data points approximately adhered to a diagonal line, resembling a normal distribution. While a small sample size can often raise concerns, the visual confirmation provides us with confidence in using the mean time as a representative measure for comparison with other studies in the field. In a study comparing the performance of nine smart contract analysis tools across a vast

```
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 and approve omitted ...
8
9  /**  @notice postcondition allowance[_owner][_spender] == remaining */
10 function allowance(address _owner, address _spender) public view returns (uint remaining);
11
12 /** @notice postcondition ((balances[from] == __verifier_old_uint(balances[from]) - value && from !=
       to) || (balances[from] == __verifier_old_uint(balances[from]) && from == to) && ok) || !ok
13 * @notice postcondition ((balances[to] == __verifier_old_uint(balances[to]) + value && from != to) ||
       (balances[to] == __verifier_old_uint (balances[to]) && from == to) && ok) || !ok
14 * @notice postcondition allowance[from][msg.sender] == __verifier_old_uint(allowance[from][msg.sender
       ]) - value || from == msg.sender
15 * @notice postcondition allowance[from][msg.sender] <= __verifier_old_uint(allowance[from][msg.sender
       ]) || from == msg.sender
16 * @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*/
17 function transferFrom(address from, address to, uint256 value) public returns (bool ok);
18 }
```

Fig. 17: ERC20 reduced specification.

dataset of 47,518 contracts [23], it was revealed that, on average, these tools required approximately 4 minutes and 31 seconds for the analysis of each contract. Remarkably, even the fastest tools spent at least 5 seconds per contract, which is slower than the average execution time of our tool ($\tilde{4}$ seconds). With this context, it is fair to assume that our tool has a reasonable execution time.

In the following, we use some snippets extracted from ERC20 examples 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 17 presents a reduced specification, focusing on functions `transferFrom` and `allowance` for the purpose of this discussion,

```
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. 18: Buggy ERC20 transferFrom function.

```
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. 19: Buggy ERC20 allowance function.

derived from the informal description in the standard [70]. 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 smart contract so it has

```
contract ERC1155 {

    mapping (uint256 => mapping(address => uint256)) private _balances;
    mapping (address => mapping(address => bool)) private _operatorApprovals;

    //... functions balanceOf, setApprovalForAll, isApprovedForAll, safeBatchTransferFrom omitted ...

    /** @notice postcondition batchBalances.length == accounts.length
    * @notice postcondition batchBalances.length == ids.length
    * @notice postcondition forall (uint x) !( 0 <= x &&  x < batchBalances.length ) || batchBalances[
        x] == _balances[ids[x]][accounts[x]] */
    function balanceOfBatch( address[] memory accounts, uint256[] memory ids ) public view returns (
        uint256[] memory batchBalances) {}

    /** @notice postcondition to != address(0)
    * @notice postcondition _operatorApprovals[from][msg.sender] || from == msg.sender
    * @notice postcondition __verifier_old_uint ( _balances[id][from] ) >= amount
    * @notice postcondition _balances[id][from] == __verifier_old_uint ( _balances[id][from] ) -
        amount
    * @notice postcondition _balances[id][to] == __verifier_old_uint ( _balances[id][to] ) + amount
    * @notice emits TransferSingle  */
    function safeTransferFrom(address from, address to, uint256 id, uint256 amount, bytes memory data
        ) public {}
}
```

Fig. 20: ERC1155 reduced specification.

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 12 to 13) require that the balances are updated as expected, whereas the purpose of the last two (Lines 14 to 15) is to ensure that the tokens available for withdrawal have been properly updated.

We use the snippet in Figure 18 — extracted from the Uniswap repository, commit 55ae25, we underline the commits that we refer to in Table 1 as well to help readers locate it more easily — 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 17 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 19 — DigixDao repository, commit 5aee64 — does not conform to its formal specification. The correct allowance for

```
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. 21: Buggy ERC1155 safeBatchTransferFrom function.

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.

The ERC1155 was created in order to promote a better integration between the ERC20

```
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. 22: safeTransferFrom function before refactoring.

```
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. 23: Successful refactoring of the safeTransferFrom function.

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 function `isApprovedForAll` returns a boolean value informing if an address is allowed to handle the tokens from another address. The `safeTransferFrom` function transfers tokens in a safe way to a valid ERC1155 address. The `transferFrom` function can do batch operations, transferring tokens to several wallets at the same time, reducing transaction costs and minimising impacts on the network. The `setApprovalForAll` function gives an address permission to handle another address' tokens.

Figure 20 presents a reduced specification for the ERC1155 standard. In the `balanceOfBatch` function, the size of the list of accounts and token ids should be equal to the list of balances returned, besides we must also check if the balances are in the list (Lines 8 to 10). The postconditions defined for the function `safeTransferFrom` (Lines 13 to 17) for the operation to be successful, the token should be debited from the source account and credited in the destination account.

The snippet in Figure 21 — extracted from Descentralized-Stock repository, commit 96d5b2 — 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 an error or an unexpected behavior. A possible solution would consist of changing the operator `!=` for `==` in the second `requires` in Figure 21.

Figures 22 and 23 — 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, 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.

Lending and credit is one of the most important financial activities and have been an integral part of human society and it is intricately related to the concept of trust and the promise of repayment [76]. The ERC3156 standard is composed by `ERC3156FlashBorrower` and `ERC3156 FlashLender` interfaces and together they provide a standardisation for single-asset flash loans, an emerging service in the decentralised finance ecosystem, allows users to request a non-collateral loan [72]. Figure 24 presents a specification for ERC3156 standard. The postcondition (Line 10) defined for `maxFlashLoan` function checks if the amount of currency available to be lent is returned correctly, according to the specification defined for this function the zero value must be returned if the token passed as parameter is not supported. The `flashFee` function must return the fee charged for a given loan (Line 14) when the token is not supported the operation must revert so when the

```
contract ERC3156FlashBorrower {

    /** @notice postcondition initiator == msg.sender  */
    function onFlashLoan(address initiator,address token,uint256 amount,uint256 fee,bytes calldata
        data) external returns (bytes32);
}

contract ERC3156FlashLender {

     /** @notice postcondition resp == line && token == address(dai) || resp == 0 */
    function maxFlashLoan(address token) external view returns (uint256 resp);

    /** @notice postcondition token == address(dai)
     *  @notice postcondition resp == (amount * toll)/WAD */
    function flashFee(address token, uint256 amount) external view returns (uint256 resp);

    /** @notice postcondition token == address(dai)
     *  @notice postcondition amount <= line
     *  @notice postcondition __verifier_old_address(token) == token
     *  @notice postcondition __verifier_old_uint(amount) == amount
     *  @notice postcondition resp */
    function flashLoan(IERC3156FlashBorrower receiver,  address token, uint256 amount,  bytes calldata
         data) external returns (bool resp);
}
```

Fig. 24: ERC3156 specification.

```
function flashLoan(IERC3156FlashBorrower
    receiver, address token, uint256 amount,
    bytes calldata data) external override lock
     returns (bool) {
        require(token == address(dai), "DssFlash
            /token-unsupported");
        require(amount <= line, "DssFlash/
            ceiling-exceeded");

        uint256 rad = mul(amount, RAY);
        uint256 fee = mul(amount, toll) / WAD;
        uint256 total = add(amount, fee);

        vat.suck(address(this), address(this),
            rad);
        daiJoin.exit(address(receiver), amount);

        emit FlashLoan(address(receiver), token,
             amount, fee);

        require(receiver.onFlashLoan(msg.sender,
             token, amount, fee, data) ==
            keccak256("ERC3156FlashBorrower.
            onFlashLoan"), "IERC3156: Callback
            failed");

        dai.transferFrom(address(receiver),
            address(this), total);
        daiJoin.join(address(this), total);
        vat.heal(rad);
        vat.move(address(this), vow, mul(fee,
            RAY));
    }
```

Fig. 25: Buggy flashLoan function.

verification process is successful it means that this condition has been met (Line 13). The `flashLoan` function initiate a flash loan and must not modify the token and amount parameter received

```
    function flashFee(address token, uint256
        value) external view returns (uint256) {
        return value.mul(9).div(10000);
    }

    function maxFlashLoan(address token)
        external view returns (uint256) {
        LendingPoolLike.ReserveData memory
            reserveData = lendingPool.
            getReserveData(token);
        return IERC20(reserveData.aTokenAddress)
            .balanceOf(address(lendingPool));
    }
```

Fig. 26: Buggy flashFee and maxFlashLoan functions.

(Lines 19 to 20). Flash lenders can provide loans of several token types on the same contract so should be checked whether or not a token is supported (Line 17) as well as the number available to lend (Line 18). If successful, the function must return true (Line 21). The `onFlashLoan` function receive a flash loan and require an initiator which should be the same as the passed as parameter by the lender function (Line 4).

The function in Figure 25 — Dss Flash repository, commits f2ca83, b1e01d and 2e70bb — does not conform the requirements defined by ERC3156 standard. According to the function signature a boolean value must be returned. After formally verifying the code on the three mentioned commits, it was detected that no explicit return value

0xMonorepo Repository											
Commit	Time	Output	Commit	Time	Output	Commit	Time	Output	Commit	Time	Output
7d59fa	3.05s	WOP	bb4c8b	2.20s	No errors	89abd7	3.18	No errors	99fbf3	2.90s	No errors
7008e8	3.25s	WOP	897515	3.11s	No errors	ba1485	3.47s	No errors	9b521a	3.45s	No errors
b58bf8	2.93s	WOP	32fead	3.78s	No errors	f21b04	3.32s	No errors	0758f2	3.73s	No errors
548fda	2.85s	WOP	d11811	4.01s	No errors	d2e422	3.51s	No errors	d35a05	3.56s	No errors
6f2cb6	3.70s	No errors	1729cf	3.28s	No errors	a2024d	2.70s	No errors	5813bb	4.20s	No errors
145fea	3.10s	No errors	63abf3	3.44s	No errors	bb3c34	3.41s	No errors	01aeee	3.21s	No errors
1fb643	3.83s	No errors	c84be8	3.20s	No errors	f54591	3.97s	No errors	272125	3.67s	No errors
5198c5	2.50s	No errors	8bce73	2.79s	No errors						

Table 3: 0xMonorepo Commit History.

had been defined by the developer, so the *false* value is returned by default. According to the function's specifications, the value *true* must be returned whenever its execution is successful. This represents a serious flaw because from the point of view of a client that invokes the `flashLoan` function, its execution would never be successful.

The code snippet in Figure 26 — Weth10 repository, commits 34c42c and b85345 — illustrates two functions that have an error when checked against its formal specification. The ERC3156 Flash Loans standard can handle multiple tokens, so it is mandatory for these functions to check the address passed by parameter and return right value or revert when it is necessary.

```
/** @notice invariant _totalSupply ==
    __verifier_sum_uint(balances) */
contract ERC20Token {

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

    // functions approve, transfer and
        transferFrom omitted ..

    /** @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. 27: Merged contract.

4.1.3 Trusted deployer tool

After discussing the results of the smart contract verification process, we introduce a scenario based on 0xMonorepo repository which implements the ERC20 pattern and is one of the repositories used during our study. We analysed the whole commit history (see Table 3) in order to find errors or nonconformities and describe what the history of the repository would be if a developer had used our tool since invalid commits would be prevented from being deployed. The idea is to show how our tool implements the architecture depicted in Figure 3; we also demonstrate the tool supports a safe smart contract development process.

As already explained, the trusted deployer 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 uses the *solc-verify* in background to verify the code against its specification before it proceeds to deploy the smart contract. Figure 27 presents a merged contract, which is the result of the merging of the specification and implementation contracts. The verification contract is automatically created from the abstract syntax tree of the contracts after a syntactic check is performed; in this case, the goal is to analyse if there is any discrepancy between the signature of the functions and the data model of the specification and implementation contracts. Provided the syntactic analysis is successful, the tool invokes the *solc-verify* in background to carry out verification of conformance to the specification.

During the analysis of the scenario, 30 commits were verified; it was observed the WOP error in the first four commits 7d59fa, 7008e8, b58bf8

ERC20, ERC721							
Repository	Commit	Time	Output	Repository	Commit	Time	Output
Bancor	4176bb	5.12s	No errors	OpenZeppelin ERC721	07603d	7.10s	No errors
Bancor	a7df76	4.45s	No errors	OpenZeppelin ERC721	09734e	8.26s	No errors
DigixDao	0a9709	6.07s	IOU	OpenZeppelin ERC721	b7d60f	7.43s	IOU
DigixDao	0550e8	7.10s	IOU	OpenZeppelin ERC721	bd0778	6.50s	No errors
DigixDao	6c717c	6.51s	No errors	Uniswap ERC20	55ae25	6.15s	No errors
DigixDao	83ad3e	6.51s	No errors	DigixDao	0390d2	5.37s	No errors
DigixDao	5571f9	6.51s	No errors	DigixDao	e320a2	7.17s	No errors
Token ERC20	5caa1d	5.42s	WOP	Uniswap ERC20	7417b2	8.09s	No errors
Token ERC20	2d7a81	4.29s	No errors	Uniswap ERC20	e382d7	7.35s	No errors
Token ERC20	44c3a1	5.33s	WOP	Set-Protocol	e85133	7.30s	No errors
Set-Protocol	23e484	5.10s	No errors	Set-Protocol	187f9b	4.62s	No errors
Set-Protocol	5999fc	5.33s	No errors	Set-Protocol	573434	4.89s	No errors
Set-Protocol	ae657d	6.45s	No errors	Set-Protocol	b1cc53	6.84s	No errors
Set-Protocol	be95aa	4.37s	No errors	Set-Protocol	bf85e7	7.15s	No errors
Set-Protocol	cad51f	8.19s	No errors	Set-Protocol	e63ae2	5.39s	No errors

Table 4: Changes of Data Representation and Interface Extension.

and 548fda. If the developer had used our tool, the error would have been discovered in the first analysis, these deployments would have been prevented, and an error message informing the specific reason would be presented to the developer, forcing him to fix the bug. From the results collected from our evolution scenario, one can see that our strategy is effective in identifying errors in the early stage of the process. Our tool abstracts many details of the deployment and upgrade process, making it simpler for platform users when compared to the manual process.

4.2 Evolution Involving Data Refinement and Interface Extension

The presentation of our case studies considering data representation and interface evolution follows the same structure as the previous case studies: first, we describe the context in Section 4.2.1 and then we provide the results in Section 4.2.2.

4.2.1 Context

To illustrate and evaluate the proposed method described in Section 3.1.1, we formally specify and analyse multiple implementations of two Ethereum smart contract specifications: ERC20 and ERC721. Unlike the previously discussed ERC20, the ERC721 [25] introduces a standard for Non-Fungible Tokens (NFTs) that are identified by a unique key. It can be useful, for instance, to distinguish collectible items, numbered seats, and so on. The specifications for each ERC are expressed under the terms of the standard RFC 2119, which defines several keywords that help describe the requirements, guide the understanding of it and extract the formal specifications. Despite the standard RFC 2119 being quite efficient for requirements management, there is no consensus regarding the data model that should be adopted for each implementation. Also, the data model or requirements (possibly including new functions) can be changed to suit specific demands. We consider scenarios that contemplate both data model changes and interface extension.

4.2.2 Results

Table 4 shows the complete list of all results we obtained. We use the same execution environment as for the previous case study (Sec. 4.1). These figures account for checking both implementation conformance and specification refinement.

To better illustrate how our strategy tackles changes in the data structure or interface, in the following, we examine four eminent commits from different open-source projects. Such commits deliberately change variable types, structs, or interfaces. In this context, the specification (against which the implementation must be verified) must also evolve, since postconditions are expressed in terms of the smart contract attributes, with their associated types. Therefore,

```
contract ERC721 is ERC165 {

    /**Mapping from owner to number of owned
        token*/
    mapping (address => uint256) private
        _ownedTokensCount;

    function balanceOf(address owner) public
        view returns (uint256 balance) {
        require(owner != address(0));
        return _ownedTokensCount[owner];
    }
}
```

(a) Original implementation.

```
contract ERC721 is ERC165, IERC721 {

    using Counters for Counters.Counter;

    /**Mapping from owner to number of owned
        token*/
    mapping (address => Counters.Counter)
        private _ownedTokensCount;

    function balanceOf(address owner) public
        view returns (uint256 balance) {
        require(owner != address(0));
        return _ownedTokensCount[owner]
            .current();
    }
}
```

(b) New implementation (commit 07603d).

Fig. 28: OpenZeppelin/ERC721 implementation delta.

a fixed (reference) specification, as in the context of the simpler scenarios presented in Section 4.1, is not suitable for the more flexible implementation evolution we address here.

First, we show how one can explicitly evolve a specification to match a new data representation. Second, we show how a new specification can be derived from an abstraction function. Third, we discuss an update not only of data representation but of the interface as well. At last, we show a simple data model update to illustrate the migration strategy for the persistent state of deployed contracts.

We begin by analyzing the commit history of the OpenZeppelin/ERC721 repository. Our purpose is to check whether changes in the data model can affect smart contract behavior.

The snippets in Figure 28 show the change in the `_ownedTokensCount` representation and the

```
contract ERC721 is ERC165, IERC721 {

    /**@notice postcondition _ownedTokensCount
        [owner] == balance*/
    function balanceOf(address owner) public
        view returns (uint256 balance) {}

    /**@notice postcondition ((
        _ownedTokensCount[from] ==
        __verifier_old_uint (_ownedTokensCount
        [from]) - 1 &&  from != to)    (from
        == to ))
    * @notice postcondition ((
        _ownedTokensCount[to] ==
        __verifier_old_uint (_ownedTokensCount
        [to]) + 1 && from != to)    (from  ==
        to))
    * @notice postcondition _tokenOwner[tokenId
        ] == to*/
    function transferFrom(address from, address
        to, uint256 tokenId) public {}
}
```

(a) Original specification.

```
contract ERC721 is ERC165, IERC721 {

    /**@notice postcondition _ownedTokensCount
        [owner]._value == balance*/
    function balanceOf(address owner) public
        view returns (uint256 balance) {}

    /**@notice postcondition ((
        _ownedTokensCount[from]._value ==
        __verifier_old_uint(_ownedTokensCount[
        from]._value) - 1 && from != to)    (
        from == to ))
    * @notice postcondition ((
        _ownedTokensCount[to]._value ==
        __verifier_old_uint (_ownedTokensCount
        [to]._value ) + 1 && from != to)    (
        from == to))
    * @notice postcondition _tokenOwner[tokenId
        ] == to*/
    function transferFrom(address from, address
        to, uint256 tokenId) public {}
}
```

(b) New specification.

```
forall (address addr) _ownedTokensCount[addr] ==
    _ownedTokensCount[addr]._value
```

(c) Abstraction function

Fig. 29: OpenZeppelin/ERC721 spec (delta).

corresponding implementation changes required to take into account the new data representation. The `Counter` struct is a wrapper for the variable `_value` and provides auxiliary methods for incrementing, decrementing, and getting the current

```
contract TokenInterface {
    struct User {
        bool locked;
        uint256 balance;
        uint256 badges;
        mapping (address => uint256) allowed;
    }
    mapping (address = User) users;
    mapping (address => uint256) balances;
    mapping (address => mapping (address =>
        uint256)) allowed;
    mapping (address => bool) seller;
}
contract Token is TokenInterface {
   function transfer(address _to, uint256 _value
       ) public returns (bool success)  {
     if (users[msg.sender].balance >= _value &&
          _value > 0) {
       users[msg.sender].balance -= _value;
       users[_to].balance += _value;
       //...
      }
      //...
   }
}
```

(a) Original implementation.

```
contract TokenInterface {
    mapping (address => uint256) balances;
    mapping (address => mapping (address =>
        uint256)) allowed;
    mapping (address => bool) seller;
}
contract Token is TokenInterface {
   function transfer(address _to, uint256 _value
       ) public returns (bool success) {
     if (balances[msg.sender] >= _value &&
          _value > 0) {
       balances[msg.sender] -= _value;
       balances[_to] += _value;
       //...
      }
      //...
    }
}
```

(b) New implementation (commit 6c717c).

Fig. 30: DigixDao/ERC20 implementation delta.

value. The idea behind this refactoring is to save gas by skipping overflow checks since it is supposedly impossible, in practice, to overflow a 256-bit integer with increments of one.

The specification also needs to be updated to reflect the changes in the data representation of the new implementation. Hence, one can explicitly generate the specification as in Figure 29 to mirror the data representation evolution. The explicit refactoring is trivial since it is just a wrapper for the value.

```
/** @notice  postcondition ( ( users[msg.sender
    ].balance ==  __verifier_old_uint (users[
    msg.sender].balance ) - _value  && msg.
    sender  != _to )      ( users[msg.sender].
    balance ==  __verifier_old_uint ( users[msg
    .sender].balance ) && msg.sender  == _to )
    &&  success )     !success */
/** @notice  postcondition ( ( users[_to].
    balance ==  __verifier_old_uint ( users[_to
    ].balance ) + _value  && msg.sender  != _to
     )      ( users[_to].balance ==
    __verifier_old_uint ( users[_to].balance )
    && msg.sender  == _to ) &&  success )
    !success */
```

(a) Original specification.

```
/** @notice  postcondition ( (
    balances[msg.sender] ==  __verifier_old_uint (
    balances[msg.sender] ) - _value  && msg.sender
     != _to )      ( balances[msg.sender] ==
    __verifier_old_uint ( balances[msg.sender] ) &&
     msg.sender  == _to ) &&  success )     !
    success */
/** @notice  postcondition ( ( balances[_to] ==
    __verifier_old_uint ( users[_to] ) + _value
    && msg.sender  != _to )      ( balances[_to]==
     __verifier_old_uint ( balances[_to] ) && msg
    .sender  == _to ) &&  success )     !
    success */
```

(b) Derived specification.

```
forall (address addr) users[addr].balance ==
    balances[addr] && users[addr].allowed ==
    allowed[addr]
```

(c) Abstraction function.

Fig. 31: DigixDao/ERC20 specification (delta).

One of the proof obligations is to check whether the generated specification is indeed a refinement of the original one. This requires that an abstraction function be provided, as explained in Section 3.1.1. In this case, the abstraction function states that, for any address `addr`, the value formerly stored as primitive data directly into the mapping now corresponds to the wrapped `_value` at the same address. In this case, the specification refinement holds.

The second proof obligation is to check whether the implementation in Figure 28 (b) conforms to the specification in Figure 29 (b). As in Section 4.1, a merged contract is generated and submitted to the *solc-verify* tool. The outcome is that there is a potential overflow. Nevertheless, as

```
/** @notice invariant totalSupply_ ==
    __verifier_sum_uint(balanceOf_)*/
contract ERC20 is IERC20 {
    uint256 public totalSupply;
    mapping (address => uint256) public
        balanceOf;
}
```

(a) Original implementation.

```
/** @notice invariant totalSupply_ ==
    __verifier_sum_uint(balanceOf_)*/
contract ERC20 is IERC20 {
    uint256 public totalSupply;
    mapping (address => uint256) public
        balanceOf;

    function mint(address to, uint256 value)
        internal {
        totalSupply = totalSupply.add(value);
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(address(0), to, value);
    }
}
```

(b) New implementation (commit e5b8db).

Fig. 32: Uniswap ERC20 new method.

```
contract TokenInterface {}
contract Token is TokenInterface {}
```

(a) Original implementation.

```
1  contract TokenInterface {
2      uint256 public totalBadges;
3  }
4
5  contract Token is TokenInterface {
6    function mintBadge(address _owner,
         uint256 _amount)
7        ifSales returns (bool success) {
8      totalBadges += _amount;
9      users[_owner].badges += _amount;
10     return success;
11   }
12 }
```

(b) New implementation (commit 0550e8d).

Fig. 33: DigixDAO new method and attribute trigger IOU.

already explained, an overflow of a 256-bit integer with increments by one will not happen in practice.

We now analyse the commit history of the DigixDao/ERC20 repository. As shown in Figure 30, a wrapper for describing user information (see the struct `User` in the original implementation) is deleted. The mapping `users` (from an address to the `User` wrapper) is also removed. In the new implementation, user information can only be obtained by direct calls to separate mappings. Unlike providing an explicit specification evolution, as in Figure 29, we illustrate here that it is possible to calculate such a specification evolution from an abstraction function (see Figure 31 (c)). The abstraction function is concretely defined. For any address `addr`, it maps the old data about balance, which is wrapped in the `User` struct, into the new corresponding special mapping `balances`. The same relationship is established for the other required information, such as `allowed`. From this abstraction function, it is possible to calculate the evolved specification, as presented in Figure 31 (b). The advantage of deriving the specification is that the specification refinement trivially holds, as in this case. The new implementation in Figure 30 conforms to this derived specification, as checked by *solc-verify*. Currently, our tool does not yet automate this calculation, but this is in our agenda for future work.

Regarding interface refinement, we discussed before that interface extension is allowed, provided the smart contract invariant is preserved. In the case of Figure 32 from repository Uniswap/ERC20, we illustrate the addition of an extra function `mint` in commit e5b8db. This function is responsible for generating new tokens, thus increasing the total supply.

Even though the `mint` function is the only extension to the original ERC20, it modifies `totalSupply` and `balanceOf`, which are both referenced in the invariant. For that reason, the new contract needs to be verified. In this case, the invariant is preserved and the contract can be safely deployed.

Another case of a change of interface and data representation is illustrated in Figure 33. This commit (0550e8d) from repository DigixDAO, however, in contrast to the previous example, may trigger an IOU error and is identified by our verification mechanism. This can be easily fixed by replacing the Lines 8-9 to use the SafeMath library, which wrappers arithmetic operations to

```
contract SetToken is ERC20, RC20Detailed {
    struct Component {
        address address_;
        uint256 unit_;
    }

    Component[] public components;
}
```

(a) Original implementation.

```
contract SetToken is ERC20, RC20Detailed {
    address[] public components;
    uint256[] public units;
}
```

(b) New implementation (commit b1cc53).

Fig. 34: Set-Protocol/ERC20 implementation delta.

check for overflows, as illustrated in the following snippet:

```
totalBadges = totalBadges.add(_amount);
users[_owner].badges = users[_owner].badges.add(
    _amount);
```

Finally, we show here yet another implementation of the ERC20 standard. In the refactoring observed in Figure 34, an array of type Component (which is a struct with two attributes) is replaced with two new arrays corresponding to each attribute of the former struct.

Rather than discussing the details of the verification, we focus now on the `Init` contract, which performs the data migration between the previous version of `SetToken` and the new one. The contract shown in Figure 35 presents two Storage representations: the `FacetStorageAbs`, which represents the abstract data model, and the `FacetStorageCon` describing the concrete model. The `init` function loops through each element of the abstract storage to assign the correct value to the corresponding concrete variable. The proof obligations, encoded as postconditions, are properly discharged by our verification strategy.

4.3 Summary and Threats to Validity

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 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. Smart contracts are increasingly popular, and we believe they will become a key and common element of trusted distributed systems in the future. Therefore, having a safe development process supported by our framework will help to increase the credibility of such a technology and promote its adoption.

Our search scope is limited to public GitHub repositories which might not cover noteworthy cases in private repositories. Therefore, relevant cases to show strengths and weaknesses of the framework may not have been included which could lead to an unintentionally biased study or less comprehensive than it could have been. In addition, the low number of examples could also be considered a threat, even though they are real-world examples with some of them being industry standards. Regarding data representation and interface evolutions specifically, it is worth mentioning that although refactorings can introduce errors, some of them are deliberate and count on practical infeasibility instead of formal correctness. Finally, although we use the git versioning mechanism to reason about the evolution of smart contracts through sequential commits, it does not mean that each commit was, in fact, deployed. Since refinement relations are transitive, however, even though there are intermediate undeployed versions, the deployment of a target version can be verified in the same fashion.

In this work, we do not focus on security properties and trust guarantees of our trusted deployer. Instead, we focus on the functional aspect of our framework. For instance, we do not discuss in detail how the trusted deployer and its infrastructure can be trusted, nor how to establish a secure communication channel with it. We leave a detailed discussion on all these aspects together with the mechanisms and protocols by which they can be implemented for follow-up work.

Throughout the study the differences between informal and formal requirements specification languages were noted. It is necessary to fulfill the gap between these approaches through a systematic mapping between them. Such an approach will help to systematise the process of defining invariants and postconditions and keep a high

```
contract Init {

    struct FacetStorageAbs {
        Component[] components;
    }

    struct FacetStorageCon {
         address[] public components;
        uint256[] public units;
    }

    FacetStorageAbs storage_abs;
    FacetStorageCon storage_con;

    /** @notice postcondition forall (uint i) i < 0 || i >= storage_abs.components.length ||
         storage_con.components[i] == storage_abs.components[i].address_
    * @notice postcondition forall (uint i) i < 0 || i >= storage_abs.components.length || storage_con.
         units[i] == storage_abs.components[i].unit_
    * @notice modifies storage_con */
    function init() public {
        storage_con.components.length = storage_abs.components.length;
        /**
         * @notice invariant forall (uint j) j < 0 || j >= i || storage_con.components[j] ==
             storage_abs.components[j].address_
         * @notice invariant forall (uint j) j < 0 || j >= i || storage_con.units[j] == storage_abs.
             components[j].unit_
         */
         for (uint i = 0; i < storage_abs.components.length; i++) {
            storage_con.components[i] = storage_abs.components[i].address_;
            storage_con.units[i] = storage_abs.components[i].unit_;
        }
    }
}
```

Fig. 35: Set-Protocol/ERC20 init function

level of traceability and adherence between the requirements and the functionalities.

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 [36, 67, 29] and community-proposed design patterns proposing mechanisms to upgrade smart contracts [65, 16, 56] attest this necessity. Yet, surprisingly, we could only find a few close related approaches [21, 60, 14] that try to tackle this problem.

The preliminary work in [21] 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 [60], 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 (and possibly an abstraction function) of the expected behaviour of a contract; we then formally verify the evolved contract against such a formal specification, as well as specification refinement when the specification also evolves. 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.

Azzopardi *et al.* [14] 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.

Some other papers have proposed methodologies to carry out pre-deployment patching/repairing [68, 53, 78]. 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 using techniques like symbolic execution [45, 51, 43], or static analysis [28, 66, 69, 58]. Tools operating on the level of Solidity were also proposed using techniques like modular program verification [32, 31], bounded model checking [73, 12], and deductive verification [7, 6]. 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 [48, 49] is a methodology that was originally created for specifying the behaviour of object-oriented programs [41, 38] but was also adopted in other contexts [32]. 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 [70].

Our notion of specification refinement is inspired by traditional notions of program refinement [50] and behavioural subtyping [42]. Nevertheless, it differs from these two notions in some aspects. The traditional notion of program refinement commonly employs stepwise refinement to take an abstract specification all the way into a concrete implementation. Our work

is concerned with a notion of type (and persistent state) evolution. We are not concerned with the same level of stepwise refinement that these frameworks typically employ. Instead, we rely on a notion of a type specification that can carefully evolve guided by implementation evolutions. Therefore, our work share more commonalities with behavioural subtyping [42]. Our notion of specification refinement is very similar to that of constraint-based behavioural subtyping [42]. We do not, however, impose a constraint on the history of (sub)types/refined specifications. Moreover, we are concerned as well with persistent states of smart contracts and how we can migrate them when the data representation changes.

Formal verification of smart contracts is a concern of the community. An entire blockchain network, Tezos [27], even acknowledges and supports formal verification from the ground up [17]. Nevertheless, upgradeability is not yet properly supported. Reasoning about updates/upgrades [19, 26] requires the extension of existing formal strategies to endorse introspection [34], i.e., to analyse current properties or behaviours in regard to previous versions of a smart contract, as we tackle in our approach. Additionally, a more practical issue regarding upgrades is the storage migration after changes in data representation, since the new version must import and adapt the state from the earlier version. There are different strategies to complete this migration, but it is non-trivial for most real-world scenarios because of the storage size. On Tezos, there is even a concept of upgradeable storage to aid the migration by adding compile-time type safety [3]. We deal with storage migration as an integrated aspect of our approach to safe evolution.

6 Conclusion

We propose a framework for the safe deployment and evolution of smart contracts. It allows the specification of a contract to be upgraded as long as the new specification conforms to the current one, and the implementation to be upgraded as long as it conforms to the specification; reference specification and implementations are set at deployment time. Resources optimisation is, for example, a reason for such an upgrade. While enforcing this safe evolution of contracts, our *trusted deployer* records which contracts it has verified and which specification they conform to. Participants can use this information to ascertain that they are interacting with a contract with the expected behaviour, ensuring a notion of safe execution.

Stakeholders of contracts managed by the trusted deployer are certain of the maintenance (or, possibly, strengthening) of the contract's invariant. Hence, properties shown to hold thanks to the invariant are guaranteed to hold for any safe evolution of the contract. Another distinctive feature of our approach is that we support the safe evolution of the persistent storage of smart contracts in a blockchain, when there is a change in data representation. 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 developed a prototype implementation of the trusted deployer and investigated its applicability — specially its formal verification component — to contracts implementing widely used Ethereum standards. First, we addressed scenarios that involve implementation evolution considering a fixed interface and data model; commit histories of the ERC20 Token Standard, ERC3156 Flash Loans and ERC1155 Multi Token Standard were explored. Then we addressed more elaborate scenarios that involve interface extension and data refinement, investigating the ERC20 Token Standard and the ERC721 standard for Non-Fungible Tokens. The results in both cases seem very promising. We could attest that some deployments and upgrades are safe, but also uncovered bugs in several commits.

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.

After shifting immutability from the implementation of a contract to its specification — and promoting the "code is law" to the "specification is law" paradigm — in [10], here we have raised again the level of flexibility experienced by developers of smart contracts. The new paradigm shift proposed in this paper can be called "specification refinement is law": apart from implementation evolution, it allows specifications to evolve as

well, provided a refinement notion is obeyed. 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 specifications; 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, new specifications/implementations that involve change of data representation or interface extension can be formally captured and verified by our framework.

One limitation of our current approach is its use of partial correctness notions. We are not concerned with showing that functions terminate. Thus, our framework is designed to capture *safety properties* but not so much *liveness* ones. In future work, we intend to define ways to prove function termination and capture liveness properties.

Some interesting refinement relations that deal with interface extension of process inheritance (in a process algebraic setting) are presented in [22]. The proposed relations allow the extension of functionality, whilst preserving behavioral properties. We will investigate the suitability of the relations proposed in [22] for smart contracts.

Despite defining and exemplifying all the artifacts necessary for automating our framework, we only have a partial implementation so far. For commit histories that are checked against a fixed specification, the user needs to input only such a specification. The merged contract and the integration with *solc-verify* in background are automated. For scenarios that involve data refinement, the user needs to provide a specification for each upgrade step and the respective abstraction function. We intend to improve this implementation in two main directions: generating formal specifications from the natural language descriptions of the functions of smart contracts (particularly in the case of the standards investigated here); and deriving a specification from a given specification and an abstraction function. The plan is to make it available to the community.

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] Lorentz documentation. Available at: https://wiki.tezos.com/build/smart-contracts/morley-framework/lorentz.

[4] *Solidity compiler*. Published at: https://github.com/ethereum/solidity.

[5] Chandra Adhikari. Secure framework for healthcare data management using ethereum-based blockchain technology. In *2017 Undergraduate Research and Scholarship Conference*, 2017.

[6] 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.

[7] 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.

[8] Nurzhan Zhumabekuly Aitzhan and Davor Svetinovic. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. In *IEEE Transactions on Dependable and Secure Computing*, pages 840 – 852, 2016.

[9] Pedro Antonino, Ante Derek, and Wojciech Aleksander Wołoszyn. Flexible Remote Attestation of Pre-SNP SEV VMs Using SGX Enclaves. *IEEE Access*, 11:90839–90856, 2023.

[10] Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. Specification is law: Safe creation and upgrade of ethereum smart contracts. In Bernd-Holger Schlingloff and Ming Chai, editors, *Software Engineering and Formal Methods - 20th International Conference, SEFM*

2022, Berlin, Germany, September 26-30, 2022, Proceedings, volume 13550 of *Lecture Notes in Computer Science*, pages 227–243. Springer, 2022.

[11] Pedro Antonino and A. W. Roscoe. Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity, 2020.

[12] 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.

[13] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST 2017*, pages 164–186. Springer, 2017.

[14] 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.

[15] 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.

[16] Gabriel Barros and Patrick Gallagher. EIP-1822: Universal Upgradeable Proxy Standard (UUPS). https://eips.ethereum.org/EIPS/eip-1822.

[17] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I 3*, pages 368–379. Springer, 2020.

[18] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: secure derivative contracts for ethereum. In *Financial Cryptography and Data Security - FC 2017 International Workshops*, pages 453–467. FC, 2017.

[19] Kai Brünnler, Dandolo Flumini, and Thomas Studer. A logic of blockchain updates. In *Logical Foundations of Computer Science: International Symposium, LFCS 2018, Deerfield Beach, FL, USA, January 8–11, 2018, Proceedings*, pages 107–119. Springer, 2017.

[20] Alberto Cuesta Cañada, Fiona Kobayashi, Fubuloubu, and Austin Williams. Erc-3156: Flash loans. *Ethereum Improvement Proposals*, 3156, 2020. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-3156.

[21] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. Proof-carrying smart contracts. In *Financial Cryptography Workshops*, 2018.

[22] José Dihego, Augusto Sampaio, and Marcel Oliveira. A refinement checking based strategy for component-based systems evolution. *J. Syst. Softw.*, 167:110598, 2020.

[23] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 530–541, New York, NY, USA, 2020. Association for Computing Machinery.

[24] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[25] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Erc-721: Nonfungible token standard. *Ethereum Improvement Proposals*, 721, 2018. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-721.

[26] Rustam Galimullin and Thomas Ågotnes. Coalition logic for specification and verification of smart contract upgrades. In *PRIMA 2022: Principles and Practice of Multi-Agent Systems: 24th International Conference, Valencia, Spain, November 16–18, 2022, Proceedings*, pages 563–572. Springer, 2022.

[27] LM Goodman. Tezos—a self-amending crypto-ledger white paper, 2014. Available at: https://www.tezos.com/static/papers/whitepaper.pdf.

[28] Ilya Grishchenko, Matteo Maffei, and Clara

Schneidewind. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep*, 2018.

[29] 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.

[30] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. Smart contract-based campus demonstration of decentralized transactive energy auctions. In *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference*, page 1–5. IEEE, 2017.

[31] Ákos Hajdu and Dejan Jovanović. Smt-friendly formalization of the solidity memory model. In *ESOP 2020*, pages 224–250. Springer, 2020.

[32] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *VSTTE*, pages 161–179. Springer, 2020.

[33] George T Heineman and William T Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, 5:1, 2001.

[34] Maurice Herlihy and Mark Moir. Blockchains and the logic of accountability: Keynote address. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 27–30, 2016.

[35] 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.

[36] 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.

[37] R.A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, 1985.

[38] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*, pages 175–188. Springer US, Boston, MA, 1999.

[39] Jonathan Lee, Kirill Nikitin, and Srinath Setty. Replicated state machines without replicated execution. IEEE, 2020.

[40] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[41] 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.

[42] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[43] 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.

[44] Shaoying Liu. Verifying consistency and validity of formal specifications by testing. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 896–914, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[45] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS 2016*, pages 254–269. ACM, 2016.

[46] 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.

[47] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *Financial Cryptography and Data Security. FC 2017. Lecture Notes in Computer Science*, volume 10322, page 357–375. Kiayias A. (eds), 2017.

[48] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[49] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., USA, 1st

edition, 1988.

[50] Carroll Morgan. *Programming from Specifications (2nd Ed.)*. Prentice Hall International (UK) Ltd., GBR, 1994.

[51] 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.

[52] Nick Mudge. EIP-2535: Diamonds, Multi-Facet Proxy. https://eips.ethereum.org/EIPS/eip-2535.

[53] 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.

[54] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. Systems of systems engineering: Basic concepts, model-based techniques, and research directions. *ACM Comput. Surv.*, 48(2), sep 2015.

[55] Benedikt Notheisen, Magnus Gödde, and Christof Weinhardt. Trading stocks on blocks - engineering decentralized markets. In *Designing the Digital Transformation. DESRIST 2017. Lecture Notes in Computer Science*. Hevner A. (eds), 2017.

[56] Santiago Palladino. EIP-1967: Standard Proxy Storage Slots. https://eips.ethereum.org/EIPS/eip-1967.

[57] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[58] 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.

[59] Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. EIP-1155: Token Standard. https://eips.ethereum.org/EIPS/eip-1155.

[60] 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.

[61] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. 2020.

[62] Jamsheed Shorish. Blockchain state machine representation. 2018.

[63] David Siegel. Understanding the dao attack. Available at: https://www.coindesk.com/understanding-dao-hack-journalists accessed on 25 September 2023.

[64] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 18(4):36–45, 2001.

[65] OpenZeppelin team. Proxy Upgrade Pattern. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies.

[66] 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.

[67] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. 54(7), 2021.

[68] Christof Ferreira Torres, Hugo Jonker, and Radu State. Elysium: Automagically healing vulnerable smart contracts using context-aware patching. *CoRR*, abs/2108.10071, 2021.

[69] 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.

[70] Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[71] Jan Vollmer. The biggest hacker whodunnit of the summer. Available at: https://www.vice.com/en/article/pgkzqm/the-biggest-hacker-whodunnit-of-the-summer, accessed on 25 September 2023.

[72] Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. Towards a first step to understand flash loan and its applications in defi

ecosystem. In *Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing*, page 23–28, 2021.

[73] 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.

[74] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[75] 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.

[76] Jiahua Xu and Nikhil Vadgama. From banks to defi: the evolution of the lending market. page 113–129. Springer, 2022.

[77] David Yermack. Corporate governance and blockchains. In *Review of Finance*, page 7–31, 2017.

[78] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, 29(4), September 2020.

[79] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.