Transaction Monitoring of Smart Contracts*

Margarita Capretto[1,2], Martin Ceresa[1], and César Sánchez[1]

[1] IMDEA Software Institute, Spain
[2] Universidad Politécnica de Madrid (UPM), Madrid, Spain

Abstract. Blockchains are modern distributed systems that provide decentralized financial capabilities with trustable guarantees. Smart contracts are programs written in specialized programming languages running on a blockchain and govern how tokens and cryptocurrency are sent and received. Smart contracts can invoke other contracts during the execution of transactions initiated by external users.
Once deployed, smart contracts cannot be modified and their pitfalls can cause malfunctions and losses, for example by attacks from malicious users. Runtime verification is a very appealing technique to improve the reliability of smart contracts. One approach consists of specifying undesired executions (*never claims*) and detecting violations of the specification on the fly. This can be done by extending smart contracts with additional instructions corresponding to monitor specified properties, resulting in an *onchain* monitoring approach.
In this paper, we study *transaction monitoring* that consists of detecting violations of complete transaction executions and not of individual operations within transactions. Our main contributions are to show that transaction monitoring is not possible in most blockchains and propose different execution mechanisms that would enable transaction monitoring.

1 Introduction

Distributed ledgers (also known as *blockchains*) were first proposed by Nakamoto in 2009 [16] in the implementation of Bitcoin, as a method to eliminate trustable third parties in electronic payment systems. Modern blockchains incorporate smart contracts [24,25], which are state-full programs stored in the blockchain that describe the functionality of blockchain transactions, including the exchange of cryptocurrency. Smart contracts allow us to describe sophisticated functionality enabling many applications in decentralized finances (DeFi), decentralized governance, Web3, etc.

Smart contracts are written in high-level programming languages for smart contracts, like Solidity [2] and Ligo [4] which are then typically compiled into low-level bytecode languages like EVM [25] or Michelson [1]. Even though smart

* This work was funded in part by the Madrid Regional Government under project "S2018/TCS-4339 (BLOQUES-CM)" and by a research grant from Nomadic Labs and the Tezos Foundation.

contracts are typically small compared to conventional software, writing smart contracts has been proven to be notoriously difficult. Apart from conventional software runtime errors (like underflow and overflow), smart contracts also suffer from new attack patterns [19] or from attacks towards the blockchain infrastructure itself [20]. Smart contracts store and transfer money, and are openly exposed to external users directly and through caller smart contracts. Once installed the code of the contract is immutable and the effect of running a contract cannot be reverted (the contract *is* the law).

There are two classic approaches to achieve software reliability, and there are attempts to apply them to smart contracts:

- **static techniques** using automatic techniques like static analysis [23] or model checking [18], or deductive software verification techniques [3,17,8,12], theorem proving [7,5,21] or assisted formal construction of programs [22].
- **dynamic verification**[13,6,15] attempting to dynamically inspect the execution of a contract against a correctness specification.

In this paper, we follow a dynamic monitoring technique. Monitors are a defensive mechanism where developers write properties that must hold during the execution of the smart contracts. If a monitored property fails the whole transaction is aborted. Otherwise, the execution finishes normally as stipulated by the code of the contract.

Most of the monitoring techniques inject the monitor into the smart contract as additional instructions [13,6,15], which is called inline monitoring [14]. The property to be monitored for a method of a given contract A is typically described as two parts: A_{begin}, that runs at the beginning of each call, and A_{end}, which is checked at the end. This monitoring code can inspect the storage of contract A and read and modify specific monitor variables. For example, monitors can compare the balance at the beginning and end of the invocation. However, monitors can only see the contents of A and cannot inspect or invoke other contracts. We call these monitors *operation monitors* as they allow us to inspect a single operation invocation. In this paper, we study a richer notion of monitoring that can inspect information across the running transaction, illustrated by our running example.

Running example: Flash Loans The aim of a flash loan contract is to allow other contracts to borrow balance *without any collateral*, provided that the borrowed money is repaid in the same transaction (perhaps with some interest) [10]. A simple way to specify the correctness of a flash loan contract A is by the following two informal properties:

FL-safety	*No transaction can decrease the balance of A*
FL-progress	*A request must be granted unless* **FL-safety** *is violated*

Fig. 1(a) shows a simple smart contract attempting to implement a flash loan lender. Function `lend` checks that the lender contract has enough tokens to provide the requested loan, saves the initial balance to later check that the loan has been repaid completely, and transfers the amount requested to the borrower. Upon return, `lend` checks that the loan has been paid back. Note that in this

```
contract Lender {
  function lend(address payable dest, uint amount) public {
begin require(amount <= this.balance);
      uint initial_balance = this.balance;
      dest.transfer(amount);
 end  assert(this.balance >= initial_balance);
  }
}
```

(a) A flash loan implementation attempt

```
contract Client {
  Lender l1, l2;
  function borrowAndInvest() public {
    l1.lend(100);l2.lend(200);
    invest(300);
    l1.transfer(100);l2.transfer(200);
  }
}
```

(b) A flash loan client

```
contract MaliciousClient {
  Lender l;
  function borrowAndInvest() public {
    l.lend(100);
    invest(100);
  }
}
```

(c) A malicious flash loan client

Fig. 1. Pseudocode for contracts `Lender`, `Client` and `MaliciousClient`.

smart contract every instruction except the transfer to the borrower is part of the operation monitor. In particular, checking that the balance is enough and saving its value is what we call A_{begin} while checking that the loan has been repaid is A_{end}.

Unfortunately, the lender smart contract in Fig. 1(a) does not fulfill property **FL-progress**. Consider a client, for example Fig. 1(b), that borrows money from different lenders, then invests the borrowed money to obtain a profit and finally pays back to the lenders. In other words, the contract `Client` in Fig. 1(b) collects all the money upfront *before* investing it and then pays back the lenders. The contract `Client` will not successfully borrow from the lender in Fig. 1(a), because contract `Lender` expects to be paid back within the scope of method `lend`. However, the contract `Client` exercises correctly **FL-safety** and **FL-progress**, and returns the borrowed tokens before the transaction finishes. The problem is that contract `Lender` is too *defensive* and only allows repayments within the control flow of function `lend` and *not in arbitrary points within the enclosing transaction*. Alternatively, a lender contract could lend funds with the hope that

```
contract Lender {
  function lend(address payable dest, uint amount) public {
    require(amount <= this.balance);
    dest.transfer(amount);
  }
} with monitor {
  uint initial_balance;
  init { initial_balance = this.balance; }
  term { assert(this.balance >= initial_balance); }
}
```

Fig. 2. A correct flash loan implementation using transaction monitors

the client returns the loan before the end of the transaction, but then a malicious contract, like in Fig. 1(c), would violate **FL-safety** easily. We cannot solve this problem with operation monitors because both A_{begin} and A_{end} are executed inside `lend` and it is not possible within the scope of `lend` to successfully predict or guarantee whether the loan will be repaid within the transaction.

In this article, we propose to extend monitors with two additional functions: A_{init}, which executes before the first call to A in a given transaction; and A_{term}, which executes after the last call to A (equivalently, at the end of the transaction). As for A_{begin} and A_{end}, A_{init} and A_{term} have access to the storage and can fail but cannot be called from other contracts or emit operations. Both A_{begin} and A_{end} can be injected into the smart contract code as additional instructions, and therefore, are executed at every invocation of A. On the contrary A_{init} and A_{term} are special functions that are invoked (by the runtime system in charge of executing the smart contracts) at the beginning and at the end of every transaction in which A is called, respectively. We call these monitors *transaction monitors* since they can check properties of the whole transaction. With transaction monitors, we implement a lender contract that satisfies **FL-safety** and **FL-progress** by saving the balance at the beginning of a transaction in `init` and comparing it with the final balance in `term` as shown in Fig. 2.

As for future work, we envision even more sophisticated monitors that guarantee properties that involve two or more contracts—like checking that the combined balance of A and B does not decrease—or even that predicating about all contracts participating in a transaction of the whole blockchain. We refer to them as *multicontract monitors* and *global monitors*, respectively, but they are out of the scope of this paper, where we focus on *transaction monitors*. Fig. 3 shows the monitoring hierarchy.

Global monitors	future work
Multicontract monitors	future work
Transaction monitors	this paper
Operation Monitors	[6,13,15]

Fig. 3. Monitors hierarchy

In summary, the contributions of the paper are the following:

– The notion of transaction monitors and its formal definition.

- A proof that current blockchains cannot implement transaction monitors, and a list of simple mechanisms that allow their implementation.
- An exhaustive study of how the proposed mechanisms interact with each other and the basic building blocks to implement full-fledged transaction monitors.

The rest of the paper is organized as follows. Section 2 describes the model of computation. Section 3 studies transaction monitors. Section 4 introduces new execution mechanisms, and in Section 5, we study how these new mechanisms implement transaction monitors. Finally, Section 6 concludes.

2 Model of Computation

We introduce now a general model of computation that captures the evolution of smart contract blockchains.

An Informal Introduction. Blockchains are a public incremental record of the executed transactions. Even though several transactions are packed in "blocks"—which are totally ordered—, transactions within a block are also totally ordered. Therefore, we can interpret blockchains as totally ordered sequences of transactions.

Transactions are in turn composed of a sequence of operations where the initial operation is an invocation from an external user. Each operation invokes a destination contract (where contracts are identified by their unique address). Operations also contain the name of the invoked method, arguments and balance (in the cryptocurrency of the underlying blockchain), and an amount of gas[3]. The execution of an operation follows the instructions of the program (the smart contract) stored in the destination address.

Given the arguments and state of the blockchain, the code of every smart contract is *deterministic* which makes the blockchain predictable and amenable to validation. We model smart contracts as pure computable functions taking their input arguments and the current local storage of the contract, and returning (1) the changes to be performed in the local storage; (2) a list of further operations to be executed. No effect takes place in their local storage until the end of the operation. This abstraction does not impose any restriction since every imperative program can be split into a collection of basic pure code blocks separated by the instructions with effects.

The execution of a transaction consists of iteratively executing pending operations, computing their effects (including updating the pending operations) until either (1) the queue of pending operations is empty, or (2) some operation fails or the gas is exhausted. In the former case, the transaction commits and all changes are made permanent. In the latter case, the transaction aborts and no effect takes place (except that some gas is consumed).

[3] The notion of gas is introduced to make all operations terminate because each individual instruction consumes gas and once the initial operation is invoked no more gas can be added to the transaction.

Model of Computation. We now formally model the state of a blockchain during the execution of the operations forming a transaction. We represent a blockchain configuration as a pair (Σ, Δ) where:

Blockchain state Σ is a partial map between addresses and the storage and balance of smart contracts,

Blockchain context Δ contains additional information about the blockchain, such as block number, current time, amount of money sent in the transaction, etc.

Blockchain contexts may vary since different blockchains carry different information, but either implicitly or explicitly, every blockchain maintains a blockchain state. The computation of a successful transaction begins with an external operation o from a configuration (Σ, Δ) and either aborts or finishes into a final configuration (Σ', Δ').

We model a *smart contract* as a partial map $A : \Delta \times \mathbb{P} \times \mathbb{S} \times \mathbb{N} \rightharpoonup (\mathbb{S} \times [\mathcal{O}])$ where $\mathbb{P}$ is the set of all possible parameters of A, $\mathbb{S}$ the set of all possible storage states, $\mathcal{O}$ the set of operations and $[\cdot]$ is a set operator representing lists of elements of a given set. Smart contracts written in imperative languages with effects can be modeled as sequences of pure blocks where effects happen at the end in the standard way.

Operations. An operation is a record containing the following fields:

- dest the address to invoke;
- src the address initiating the operation;
- param parameters expected by the smart contract at address dest;
- money the amount of crypto-currency sent in the operation.

We use standard object notation to access each field, so o.dest is the destination address, o.src is the source address, o.param the parameters and o.money the amount transferred.

Transactions. A transaction results from the execution of a sequence of operations starting from an external operation placed by an external user. If an operation fails the transaction fails and the blockchain state remains unchanged. A successful operation o results in a new storage and a list of new operations ls. The blockchain updates the storage of smart contract o.dest and balance of both smart contracts o.dest and o.src generating a new blockchain configuration and the list ls is added to the current pending queue of operations. Operations are executed one at a time modifying the blockchain configuration until some operation fails or there is no more operations on the pending queue. In the second case, the transaction is successful and the last blockchain configuration consolidates.

We assume there is an implicit partial map from addresses to smart contracts $\mathbb{G} : \mathit{Addr} \rightharpoonup \mathit{SmartContract}$. Moreover, we assume map $\mathbb{G}$ does not change since we assume that smart contracts cannot install new contracts.

Operation Execution. Let o be an operation and (Σ, Δ) a blockchain configuration. The evaluation of o from (Σ, Δ) results in a new configuration and a list of operations ls, which we denote $(\Sigma, \Delta) \xrightarrow{o} (\Sigma', \Delta', ls)$ whenever:

1. The source smart contract has enough balance, $\Sigma(o.\mathsf{src}) \geq o.\mathsf{money}$
2. The invocation to the smart contract is successful:

$$\mathbb{G}(o.\mathsf{dest})(\Delta, o.\mathsf{param}, \Sigma(o.dst).st, \Sigma(o.dst).balance) = (st', ls)$$

The new blockchain configuration state Σ' is the result of: 1) adding $o.\mathsf{money}$ into the balance of $o.\mathsf{dest}$ and subtracting it from $o.\mathsf{src}$, and 2) updating the storage as $\Sigma'(o.\mathsf{dest}).st = st'$. Note that we leave the evolution of Δ unspecified as it is system dependant. In Section 5, we implement different additional blockchain features by inspecting (and possibly modifying) the blockchain context. For failing evaluation of operations, we use $(\Sigma, \Delta) \xrightarrow{o} \lightning$.

Execution Order. The execution can proceed in different ways. We consider two execution orders: new operations are added to the beginning of the pending queue (a DFS strategy) and new operations added to the end of the pending queue (a BFS strategy). This results in the following transition rules:

$$\frac{(\Sigma, \Delta) \xrightarrow{o} \lightning}{(\Sigma, \Delta, o :: os) \not\rightsquigarrow_a}$$

$$\frac{(\Sigma, \Delta) \xrightarrow{o} (\Sigma', \Delta', ls)}{(\Sigma, \Delta, o :: os) \rightsquigarrow_{dfs} (\Sigma', \Delta', ls +\!\!+ os)} \qquad \frac{(\Sigma, \Delta) \xrightarrow{o} (\Sigma', \Delta', ls)}{(\Sigma, \Delta, o :: os) \rightsquigarrow_{bfs} (\Sigma', \Delta', os +\!\!+ ls)}$$

The execution starting from an external operation o is a sequence of steps $(\rightsquigarrow_a)$—with a fixed to be either *dfs* or *bfs*—until the pending operation list is empty or the execution of the next operation fails. Beginning from a blockchain configuration (Σ, Δ) and an initial operation o, a transaction execution is a sequence of operation executions: $(\Sigma, \Delta, [o]) \rightsquigarrow_a (\Sigma_1, \Delta_1, os_1) \rightsquigarrow_a \ldots \rightsquigarrow_a (\Sigma_n, \Delta_n, [])$ or that $(\Sigma, \Delta, [o]) \rightsquigarrow_a (\Sigma_1, \Delta_1, os_1) \rightsquigarrow_a \ldots \rightsquigarrow_a (\Sigma_n, \Delta_n, os_n) \not\rightsquigarrow_a$

A transaction can fail either because of gas exhaustion or an internal operation has failed, and in that case, we have a sequence of $\rightsquigarrow_a$ leading to a final step marked as $\not\rightsquigarrow_a$ following the failing operation.

Finally, after every successful execution, the blockchain takes the last configuration and upgrades its global system.

The model of computation described in this section does not follow exactly a call-and-return model like the Ethereum blockchain does [25]. However, it is easy to see that it can be simulated in our model by having each contract explicitly keeping its stack of returned values.

3 Transaction Monitors

We now introduce *transaction monitors* and show that it is not possible to implement them in current blockchains. We present different extensions that allow us to implement transaction monitors.

3.1 Transaction Monitors

Transaction monitors allow us to reason about properties of transactions. Each smart contract A is equipped with a monitor storage and four especial methods A_{init}, A_{begin}, A_{end} and A_{term}. These new methods cannot emit operations or modify smart contract storage, however, they have their own monitor storage. We assume that these new methods are interpreted by the blockchain and if one of these methods fail the whole transaction fails. Otherwise, the effect in the blockchain is the same as if it was executed without monitors. The functions A_{init} and A_{term} can read the storage and balance of the smart contract and read and write the monitor storage. Function A_{init} is executed before the first time A is invoked in the transaction and function A_{term} is invoked after the last interaction to A finished in the transaction, and does not modify the monitor storage. Functions A_{begin} and A_{end} are executed at the beginning and at the end of each *operation* that is executed in A, as in operation monitors [13] (note that A_{begin} and A_{end} can be easily implemented by inlining their code around the methods of A). The method A_{begin} takes the same arguments as any A operation plus the monitor storage, while function A_{end} has access to the result of the operation (list of the operation emitted and the new storage) plus the monitor storage. We call the resulting smart contracts *monitored smart contracts*.

Operation Monitors. We first extend the model of computation to include operation monitors. A monitored operation execution is a normal operation execution where the corresponding operation monitor is executed before and after the operation is executed.

We define $(\xrightarrow[mon]{o})$ modifying $(\xrightarrow{o})$ as follows. Before executing o, (1) procedure $\mathbb{G}(o.\mathsf{dest}).\mathit{begin}$ is invoked, then (2) operation o is executed, and (3) finally $\mathbb{G}(o.\mathsf{dest}).\mathit{end}$ runs. That is, operation monitors are simply restricted functions executed before and after each operation. We can then specialize $\rightsquigarrow_a$ with operation monitors, that is, use relation $(\xrightarrow[mon]{o})$ instead of relation $(\xrightarrow{o})$ to obtain transaction executions that use operation monitors.

Procedures *begin* and *end* can only modify the private monitor storage and fail, and thus, they cannot interfere in the normal execution of smart contracts (except by failing more often).

Transaction Monitors. We redefine transaction monitors execution as a restriction of the transaction execution relation so transactions invoke *init* and *term* when required. In this case, *init* can change the monitor storage, and thus, can modify the blockchain state. We define a new relation $\twoheadrightarrow_a$ the smallest relation defined by the following inference rules:

$$\frac{A_{init}(\Sigma(A)) = \Sigma'}{(\Sigma, \Delta, os) \twoheadrightarrow_a (\Sigma', \Delta, os)} \qquad \frac{A_{term}(\Sigma(A))}{(\Sigma, \Delta, []) \twoheadrightarrow_a (\Sigma, \Delta, [])} \qquad \frac{(\Sigma, \Delta, o :: os) \rightsquigarrow_a (\Sigma', \Delta', os')}{(\Sigma, \Delta, o :: os) \twoheadrightarrow_a (\Sigma', \Delta', os')}$$

Note that we sacrifice a deterministic operational semantics in favor of a clearer set of rules. As before, we use $(\nrightarrow)$ to represent failing transactions.

$$\frac{(\Sigma, \Delta) \overset{o}{\not\rightarrow}}{(\Sigma, \Delta, os) \not\rightarrow} \quad \frac{A_{init}(\Sigma(A)) \not\rightarrow}{(\Sigma, \Delta, os) \not\rightarrow} \quad \frac{A_{term}(\Sigma(A)) \not\rightarrow}{(\Sigma, \Delta, []) \not\rightarrow}$$

Finally, we define a monitored trace of a transaction same as before, given a blockchain configuration (Σ, Δ) and an external operation o:

$$(\Sigma, \Delta, [o]) \twoheadrightarrow_a (\Sigma_1, \Delta_1, os_1) \twoheadrightarrow_a (\Sigma_2, \Delta_2, os_2) \twoheadrightarrow_a \ldots \twoheadrightarrow_a (\Sigma_n, \Delta_n, [])$$

To remove the non-determinism we add a new relation that restricts the legal runs. This relation knows the set of visited addresses (smart contracts), and invokes an initialization method, and at the very end of the evaluation of a transaction uses the same set to invoke their corresponding term method.

$$\frac{(\Sigma, \Delta, os) \twoheadrightarrow_a^o (\Sigma', \Delta', os') \qquad o.\mathsf{dest} \in E}{E \vdash (\Sigma, \Delta, o :: os) \Rightarrow_a E \vdash (\Sigma', \Delta', os')}$$

$$\frac{(\Sigma'', \Delta, os) \rightsquigarrow_a^o (\Sigma', \Delta', os') \qquad o.\mathsf{dest} \notin E \qquad (\Sigma, \Delta, os) \twoheadrightarrow_a^{A_{init}} (\Sigma'', \Delta, os)}{E \vdash (\Sigma, \Delta, o :: os) \Rightarrow_a E \cup \{o.\mathsf{dest}\} \vdash (\Sigma', \Delta', os')}$$

$$\frac{(\Sigma, \Delta, []) \twoheadrightarrow_a^{A_{term}} (\Sigma, \Delta, os) \qquad e \in E}{E \vdash (\Sigma, \Delta, []) \Rightarrow_a E \setminus \{e\} \vdash (\Sigma, \Delta, [])}$$

As result, we only accept traces generated by relation $(\Rightarrow_a)$, beginning with a blockchain configuration (Σ, Δ) and an external operation o resulting in failure or a new blockchain configuration (Σ', Δ'): $\emptyset \vdash (\Sigma, \Delta, [o]) \Rightarrow_a \ldots \Rightarrow_a \emptyset \vdash (\Sigma', \Delta', [])$.

3.2 Transaction Monitors in BFS/DFS

Unfortunately, transaction monitors cannot be implemented in blockchains that follow DFS or BFS evaluation strategies. We show now a counter-example. Consider a transaction monitor for A that fails when smart contract A is called **exactly** once in a transaction. The monitor storage contains a natural number to keep track of how many times A has been invoked in the current transaction. Function *init* sets this counter to 0, *begin* adds one to the counter, *end* does nothing, and *term* fails if the monitor storage is exactly one.

Now let (Σ, Δ) be a blockchain configuration, and let A and B be two smart contracts, where A is being monitored for the "only once" property. Consider the following two executions of external operations from (Σ, Δ):

- o_1 invokes $B.f$ which then invokes o_{A1} in A,
- o_2 invokes $B.g$ which then invokes o_{A1} and o_{A2} in A.

The monitor for "only once" must reject the transaction beginning with o_1, but accept the transaction beginning with o_2.

Consider a DFS strategy. Starting from o_1, the execution trace is

$$(\Sigma, \Delta, [o_1]) \rightsquigarrow_{dfs} (\Sigma_1, \Delta_2, [o_{A1}]) \rightsquigarrow_{dfs} (\Sigma_2, \Delta_2, as_1)$$

with corresponding sequence of pending operations $[o_1]$, $[o_{A1}]$, as_1. Starting from o_2 the sequence of pending operations is $[o_2]$, $[o_{A1}; o_{A2}]$, $as_1 +\!\!+ [o_{A2}]$,...,$[o_{A2}]$, as_2. It is not possible to distinguish between the traces generated by o_1 and o_2, as anything that operation o_{A1} and its descendants as_1 do will happen before the execution of o_{A2} in the second transaction. In other words, o_{A1} and all the operations that can be generated by it or its descendants cannot know that some other invocation to A is pending. Therefore, A cannot fail during the execution of o_{A1} or its descendants, as this implies that also a failure in the execution of o_2. At the same time o_{A1} is the only chance in A to make the first transaction fail because there is no other operation in A. Consequently, the two runs are identical up to the end of o_{A1} but one must fail and the other must not fail.

A BFS scheduler can distinguish between the execution of operations o_1 and o_2 by using a *recurring operation*. Basically, a recurring operation is just a regular function that either terminates or reinserts itself in the pending queue of operations. Since new operations are added to the end of the pending queue, A can inject a recurring operation that check the state of A and conditionally, if the test that would make *term* fail is true, reinjects itself again. This recurring operation will be invoked at the end of all other functions in A. If the condition that makes *term* accept is never met, the transaction fails because the recurring operation injects itself ad-infinitum, exhausting gas. In Section 5.1 we use recurring operations thoroughly. However, a simple variation of this example that includes comparing with a third transaction where A is invoked three times shows that BFS cannot implement "only once" either (as BFS cannot distinguish between the third invocation to A and a first invocation to A in a transaction following the one originated by o_2). For a detailed proof see [9].

4 Execution Mechanisms

We propose new mechanisms and study if they help to implement transaction monitors. However, adding features to blockchains is potentially dangerous since it can introduce unwanted behaviour[19]. We focus on simple mechanisms that are easy to implement and are backwards compatible.

Since A_{begin} and A_{end} can already be implemented using inlining, we focus on mechanisms that allow executions at the beginning and end of transactions, which can aid to implement A_{init} and A_{term}. It is worth noting that in most modern blockchain smart contracts are normal functions that also manage tokens. That is, smart contracts can modify their local memory (storage), invoke another functions, fail and also transfer tokens. However, smart contracts are oblivious to the notion of transactions: they cannot tell if a new transaction has started, if two invocations belong to the same transaction or not, or when a transaction has finished. The mechanisms that we introduce in this Section will help to distinguish these situations.

We present two kinds of mechanisms, ones that introduce a new instruction, and others that add a new special method to smart contracts. In the next section, we compare their relative power and if they can implement transaction monitors.

Mechanisms that Add New Instructions. The first four mechanisms add new instructions and can be easily implemented by bakers/miners collecting the information required in the context Δ.

- *First.* We consider a new instruction, `first`, which returns true if the current operation is the first invocation to the smart contract in the current transaction. The context Δ can be extended to contain the set of contracts F that have already run an operation in the current transaction, which allows us to implement `first` as $A \notin F$, where A is the smart contract that executes `first`.
- *Count.* We introduce now a new instruction, `count` that returns how many invocations have been performed to methods of the contract in the current transaction. Again, the context Δ can easily count how many times each contract has been invoked.
- *Fail/NoFail.* This mechanism equips each contract with a new flag `fail` that can be assigned during the execution of the contract (and that is false by default). The semantics is that at the end of the transaction, the whole transaction would fail if some contract has the `fail` bit to true. For example, the failing bit allows us to implement flash loans as follows. A lender smart contract can set `fail` to true when is lending money and change it to false only when the money is returned.
- *Queue info.* We add a new operation, `queue`, indicating if there is no more interaction between smart contracts. Or equivalently, if the only operations permitted in the pending queue are recurrent operations (which can only inject operations to the same contract). These operations must also be specially qualified in the contract, and the runtime system must make sure that they only generate operations to the same contract.

Mechanisms that Add New Methods or Storage. The following mechanisms modify the definition of smart contracts either by adding new methods that are executed at particular moments in a transaction or by adding special storage/memory.

- *Transaction Memory.* Smart contracts are equipped with a special volatile memory segment that exists only during the execution of a transaction and which is created and initialized at the beginning of the transaction. We add a new segment in the smart contract indicating the initial values to be assigned. In concrete, each contract A indicates a new storage type for the transaction memory and a procedure that initializes it (which can read but not change the conventional storage). We use `trmem` to refer to this mechanism.
- *Storage Hookup, Bounded and Unbounded.* The idea is to equip smart contracts with a new method that updates the storage after the last local operations in the transaction. These methods can only modify the storage but

not invoke other methods. A bounded version of this mechanism is restricted to terminating non-failing functions (for example, by restricting the class of programs). In addition, the unbounded version is arbitrary code that can fail. We use `bstore` and `ustore` to refer to these mechanisms.

For space purposes correct flash loan implementations using these mechanisms are not included here but can be found in [9].

5 Implementing Transaction Monitors

We say a mechanism M *implements* another mechanism N whenever every smart contract executing in a blockchain with N can be simulated by a smart contract in a blockchain with M. Here, simulation means that all observable effect (in terms of failure behavior, storage changes and token transfers) are identical. We say that two mechanisms are *equivalent* if and only if they can implement each other. In this paper we disregard gas consumption so we implicitly assume that one can always assign sufficient gas to a contract.

Theorem 1. *The following are equivalent:* `trmem`*,* `first`*,* `count`*, and* `bstore`*.*

If contracts can know when their first invocation in the transaction occurs, they can set the storage in different ways simulating `count` and `trmem`. Also, `count` and `trmem` can simulate `first`, by checking if the count is 0 and initializing a volatile bit to *true*. More interesting is that `first` can simulate bounded storage hookup by applying the effect on the storage of bounded storage hookup at the beginning of the next transaction. Detailed proofs are included in the longer version of this paper [9].

Lemma 1. *Mechanism* `ustore` *implements* `bstore` *and* `fail`*.*

Proof. Mechanism `ustore` implements `bstore` trivially as it is just less restrictive. For `fail` we add in the storage of A a new field, fl to represent the failing bit which is initialized to false when the contract is installed and updated to simulate the `fail` instruction. At the end of the transaction, the `ustore` hookup checks if fl is true and fail. Otherwise, it does nothing. □

It can be proven that the other direction is not always possible. Fig 4 shows graphically the previous results where an arrow indicates that one mechanism implements another. In this diagram, an absence of an arrow does not necessarily imply impossibility but perhaps that the result depends on the execution order. For example, in BFS blockchains `first` can implement `ustore`, but this is impossible with DFS.

Since `first`, `count`, `bstore` and `trmem` are all equivalent, from now on we only refer to mechanism `first`. It is easy to see that this mechanism is enough to implement *init*.

To implement *term*, we can either implement `fail` or `ustore`, where `fail` is simpler, and `ustore` is more powerful but requires a bigger change to blockchains.

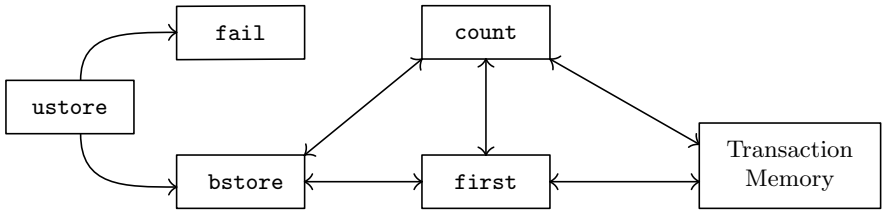

Fig. 4. Relation between mechanisms for any scheduler. An arrow from mechanism M to mechanism N means that M implements N.

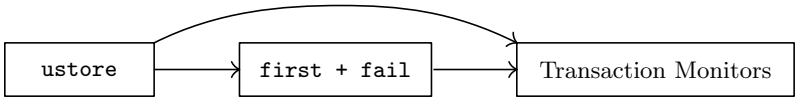

Fig. 5. Relation between mechanisms and transaction monitor for any scheduler.

Theorem 2. *Mechanisms* `first` + `fail` *implement transaction monitors.*

Proof. Let $\mathcal{B}$ be a blockchain that implements `first` and `fail`. Given a monitored smart contract A, we want to implement A in blockchain $\mathcal{B}$. We define a new smart contract A' extending its storage to also contains A's monitor storage. Then, we equip A' with a new method f' for every method f in A, such that, f' first checks `first` and executes A_{init} if needed. Then, before exiting, f' executes A_{term} with the current state but instead of failing explicitly f' set the failing bit. Function A_{init} is executed exactly once and A_{term} may be executed multiple times, but it does not modify the contract storage and it does not generate operations. The last execution of A_{term} in A' will simulate A_{term} in A. If the semantics of the blockchain were such that the balance of pending outgoing operation would subtract balance from A when it executes, then these calculations can be made in the monitor storage when the operations are generated. □

Since `ustore` implements `first` and `fail`, it follows that `ustore` implements transaction monitors.

Corollary 1. `ustore` *implements transaction monitors but transaction monitors cannot implement* `ustore`*.*

Transaction monitors can only make contracts fail but not change the storage. Our results are summarized in Fig. 5.

5.1 BFS Blockchains

We now study more in detail the mechanism for BFS based blockchains. The first result is that unless equipped with further mechanisms, BFS blockchains cannot implement transaction monitors. The essence of the proof is to create two transactions on a monitored contract A (like in "only once") in which corresponding invocations to the same contract A receive identical information, and one must fail and the other commit.

Theorem 3. *A BFS blockchain does not implement transaction monitors.*

A BFS blockchain guarantees that new operations are executed after all pending operations, which enables the implementation of `fail` using recurring operations. A recurring operation is a private function that can read and write the storage and that either terminates or reinjects itself again to the pending queue. Since every time the operation is executed the blockchain consumes gas, and eventually, failure follows from an attempt to inject itself ad-infinitum.

Lemma 2. *Recurring operations in BFS blockchain allow to implement* `fail`*.*

Since transaction monitors cannot be implemented within a BFS blockchain (see [9]), we conclude that `fail` does not implement transaction monitors in BFS blockchains. The missing element is `first` which allows to implement `ustore`. And, since `ustore` implements transaction monitors (Corollary 1), `first` can also implement transaction monitors.

Lemma 3. *Mechanism* `first` *implements* `ustore` *in BFS blockchains.*

Proof. Assume a BFS blockchain implementing `first`. Let A be a smart contract. We modify A to contain a second copy S' of its storage. Upon the first call of A, we update the current storage using the values in S'. We add a new private method *hookup* in A that mimics the code of `ustore` but (1) it applies the changes in S', and (2) instead of failing (if `ustore` fails) it calls itself as a recurring operation. Finally, we modify A so that function *hookup* is invoked at the end of each method in A. In effect, *hookup* is preventively evaluating `ustore` on the side memory S', and simulating the failure as a recurring operation (when `ustore` fails). Therefore, if the operation is the last one on the contract and it does not fail, then S' contains the correct storage, which will then be copied at the beginning of the next transaction. □

In the previous proof, we split mechanism `ustore` into two parts: one in charge of updating the storage, the other in charge of failing. If we also add `queue`, we can implement `ustore` without failing by gas exhaustion because now the *hookup* executed recurrently can know if there are only recurrent operations and then execute the `ustore` code (including the failure).

Lemma 4. *Mechanism* `queue` *implements* `ustore` *in BFS blockchains.*

In a BFS blockchain, `ustore` implements transaction monitors (Corollary 1), and thus, by the previous lemma, `queue` also implements transaction monitors. Next, we will show that `queue` cannot be implemented with `ustore` when a BFS strategy is used. Intuitively, mechanism `queue` adds a way for smart contracts to know the state of the blockchain, i.e. if there is still interaction between smart contracts, and thus, smart contracts can take different actions based on the state of the blockchain, while mechanism `ustore` adds a way to execute a procedure at the end of transactions, but smart contracts are oblivious about interactions between smart contracts. Since `ustore` implements all other mechanisms, we have that no other mechanism can implement `queue`.

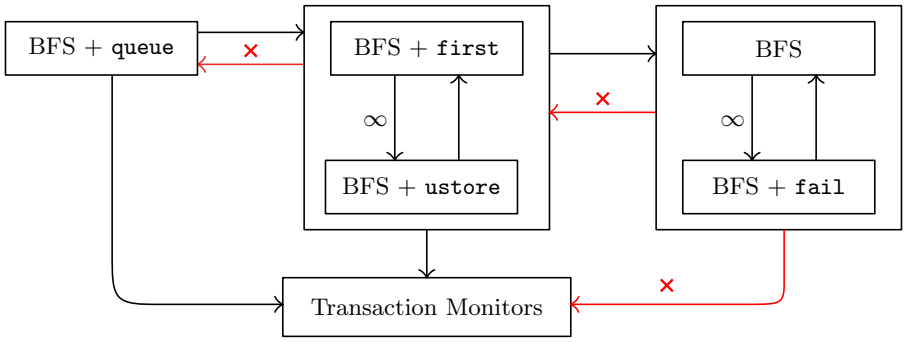

Fig. 6. Relation between mechanisms and transaction monitor in BFS blockchains. A black arrow from mechanism M to mechanism N means that M can implement N. The ∞ symbol represents the use of an infinite recursion to provoke a failure. A red arrow with a cross from mechanism M to mechanism N means that M cannot implement N.

Lemma 5. *In BFS blockchains* `ustore` *cannot implement* `queue`*.*

The main idea is to create two executions that are identical unless one can inspect the pending operation queue, and in which one operation must fail if `queue` returns that the queue of pending operations is empty. The complete proof is in [9]. Fig 6 summarizes the relations between mechanisms and transaction monitor in BFS blockchains.

5.2 DFS Blockchains

We now study DFS blockchains, that is, when the resulting list of operations from smart contracts execution are appended at the beginning of the list. This is the most conventional execution order in most blockchains, like Ethereum. We now prove several impossibility results.

Mechanisms `ustore` and `first` plus `fail` implement transaction monitors (Corollary 1 and Thoerem 2). In a DFS blockchain, those are the only two ways using our mechanisms to implement transaction monitors. We show that transaction monitors cannot be implemented by combining `queue` with either `first` or `fail`, and as a consequence none of these mechanisms on their own can implement transaction monitors.

Lemma 6. *A DFS blockchain implementing* `queue` *and* `first` *does not implement transaction monitors.*

Proof. Let $\mathcal{B}$ be a DFS blockchain and A a smart contract installed in $\mathcal{B}$. Consider the "only once" monitor that fails if and only if the smart contract A is called exactly once. We show that this monitor cannot be implemented in DFS even with `first` and `queue`.

Let B, C be two other smart contracts. We analyze the pending queue of execution of two possible external operations originated by B:

1. o_1 where B calls A *once* and then C
2. o_2 where B calls A *twice* and then C

We assume that there are no additional invocations to A aside from the described above. When we execute both operations in a (Σ, Δ) blockchain system, we have the following two traces:

- $t_1 : (\Sigma, \Delta, [o_1]) \rightsquigarrow_{dfs} (\Sigma', \Delta', [a_1, c_1]) \dots$
- $t_2 : (\Sigma, \Delta, [o_2]) \rightsquigarrow_{dfs} (\Sigma', \Delta', [a_1, a_2, c_1]) \dots$

Note that the presence of operation c_1 in the pending execution queue is forcing mechanism `queue` to return false. Since the occurrence of operation a_1 in both cases execute in the same configuration, the behavior must be the same. The transaction executing o_1 must fail because A is called only once, but this will make the second transaction fail as well. □

We can conclude that neither `queue` nor `first` alone would implement transaction monitors.

Lemma 7. *Under DFS* `queue` *and* `fail` *cannot implement transaction monitors.*

The main difference between these mechanisms and transaction monitors is that the latter can execute functions without a contract being invoked at particular moments in the execution of transactions. Take for example procedure *init*, neither `queue` nor `fail` can simulate *init*, as there is no way for these mechanisms to distinguish the first execution of a smart contract in a given transaction.

Combining `fail` with `first` one can implement transaction monitors in any execution order, including DFS (Theorem 2), but `fail` is not enough to implement transaction monitors in DFS. Therefore, we conclude that DFS blockchains do not implement `first`. Moreover, putting all previous lemmas together, we conclude that a DFS blockchains cannot implement any of the mechanisms listed in Section 4 directly.

Corollary 2. *DFS blockchains cannot implement* `first`, `fail`, `ustore` *or* `queue`.

All proofs are in [9].

6 Conclusion and Future Work

We have studied transaction monitors for smart contracts. Transaction monitors are a defense mechanism enabling smart contracts to explicitly state wanted or unwanted behaviour at the transactional level. This kind of properties are motivated by contracts like flash loans, which are not implementable in their full generality in current blockchains. We propose a solution based on adding new mechanisms to the blockchain. Transaction monitors can be incorporated directly into contracts or simulated if some of these mechanisms are implemented. This

could be preferable since some of these mechanisms are very simple and backward compatible, while others extend the functionality of smart contracts. We have studied how some mechanisms simulate each other, both for any execution order, and specifically for BFS and DFS blockchains. The conclusion is that the simplest mechanism that allows us to implement transaction monitors is the combination of `first` and `fail`.

Nevertheless, the main contribution of this paper is purely theoretical. Future work includes implementing transaction monitors and practically interesting features from Section 4 in a real blockchain and implement illustrative transaction monitors.

For simplicity, we have neglected a specific analysis of gas consumption, except for recurrent operations that purposefully fail by exhausting gas. Even though transaction monitors will consume additional gas which can influence the failure of the transaction (as with operation monitors), we claim that for all our development there is an amount of gas that can be calculated which will not make accepting transactions fail. However, we leave a detailed study for future work.

Other avenues of future work include the study of new features, particularly *views* that allows contracts to inspect the state of other contracts. We are also performing a thorough study of how exposing new mechanisms to contracts—that can use them for implementing functionality—can break (or not) implementations of monitors that are correct without adding the mechanisms. Finally, since proofs in this paper are "pencil and paper" and the interplay of different mechanisms can be counter-intuitive, we plan to formalize all proofs here in an existing smart-contract formal "playgound" (libraries in theorem provers that enable mechanical proofs), e.g. [11].

References

1. Michelson: the language of smart contracts in Tezos. `https://tezos.gitlab.io/whitedoc/michelson.html`.
2. Ethereum. Solidity documentation — release 0.2.0. `http://solidity.readthedocs.io/`, 2016.
3. W. Ahrendt and R. Bubel. Functional verification of smart contracts via strong data integrity. In *Proc. of ISoLA (3)*, LNCS, pages 9–24. Springer, 2020.
4. G. Alfour. LIGO: a friendly smart-contract language for Tezos. `https://ligolang.org`, 2020. last accessed: 2022-05-03.
5. D. Annenkov, J. B. Nielsen, and B. Spitters. ConCert: a smart contract certification framework in Coq. In *Proc. of the 9th ACM SIGPLAN Int'l Conf. on Certified Programs and Proofs (CPP'20)*, pages 215–218. ACM, 2020.
6. S. Azzopardi, J. Ellul, and G. J. Pace. Monitoring smart contracts: ContractLarva and open challenges beyond. In *Proc. of the 18th International Conference on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 113–137. Springer, 2018.
7. B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. Mi-Cho-Coq, a framework for certifying Tezos smart contracts. *arXiv*, abs/1909.08671, 2019.

8. K. Bhargavan, A. Delignat-Lavaud, C. Fourneta, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Z. Béguelin. Formal verification of smart contracts: Short paper. In *Proc. of Workshop on Programming Languages and Analysis for Security (PLAS@CCS'16)*, pages 91–96. ACM, 2016.
9. M. Capretto, M. Ceresa, and C. Sánchez. Transaction monitoring of smart contracts. *arXiv*, abs/2207.02517, 2022.
10. A. C. Cañada, F. Kobayashi, fubuloubu, and A. Williams. Eip-3156: Flash loans.
11. M. Ceresa and C. Sánchez. Multi: a formal playground for multi-smart contract interaction. In C. Schneidewind and Z. Dargaye, editors, *4th Int'l Workshop on Formal Methods for Blockchains, FMBC@CAV 2022 (to appear)*, OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
12. S. Conchon, A. Korneva, and F. Zaïdi. Verifying smart contracts with Cubicle. In *Proc. of the 1st Workshop on Formal Methods for Blockchains (FMBC'19)*, volume 12232 of *LNCS*, pages 312–324. Springer, 2019.
13. J. Ellul and G. J. Pace. Runtime verification of Ethereum smart contracts. In *Proc. of the 14th European Dependable Computing Conference (EDCC'18)*, pages 158–163. IEEE Computer Society, 2018.
14. M. Leucker. Teaching runtime verification. In *Proc. of RV'11*, number 7186 in LNCS, pages 34–48. Springer, 2011.
15. A. Li, J. A. Choi, and an. Long. Securing smart contract with runtime validation. In *Proc. of ACM PLDI'20*, pages 438–453. ACM, 2020.
16. S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009.
17. Z. Nehaï and F. Bobot. Deductive proof of industrial smart contracts using Why3. In *Proc. of the 1st Workshop on Formal Methods for Blockchains (FMBC'19)*, volume 12232 of *LNCS*, pages 299–311. Springer, 2019.
18. A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, 2020.
19. D. Phil. Analysis of the dao exploit, 2016.
20. D. Robinson and G. Konstantopoulos. Ethereum is a dark forest, 2020.
21. J. Schiffl, W. Ahrendt, B. Beckert, and R. Bubel. Formal analysis of smart contracts: Applying the KeY system. In *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *LNCS*, pages 204–218. 2020.
22. I. Sergey, A. Kumar, and A. Hobor. Scilla: a smart contract intermediate-level LAnguage. *CoRR*, abs/1801.00687, 2018.
23. J. Stephens, K. Ferles, B. Mariano, S. Lahiri, and I. Dillig. SmartPulse: Automated checking of temporal properties in smart contracts. In *Proc. of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*. IEEE, May 2021.
24. N. Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 16, 1996.
25. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.