Monitoring the Future 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 decentralized systems that provide trustable execution guarantees through the use of programs called smart contracts. Smart contracts are programs written in domain-specific programming languages running on blockchains that govern how tokens and cryptocurrency are sent and received. Smart contracts can invoke other smart contracts during the execution of transactions initiated by external users.
Once deployed, smart contracts running code cannot be modified, so techniques like runtime verification are very appealing for improving their reliability. Moreover, the conventional model of computation of smart contracts is transactional: once operations commit, their effects are permanent and cannot be undone. Therefore, errors in smart contracts may lead to millionaire losses of money.
In this paper, we present the concept of *future monitors* which allows monitors to remain waiting for future transactions to occur before committing or aborting. This is inspired by optimistic rollups, which are modern blockchain implementations that increase efficiency (and reduce cost) by delaying transaction effects. We exploit this delay to propose a model of computation that allows bounded future monitors. We show our monitors correct respect with legacy transactions, how they implement bounded future monitors and how they guarantee progress. We illustrate the use of bounded future monitors by implementing correctly multi-transaction flash loans.

1 Introduction

Blockchains [20] were first introduced as distributed infrastructures that eliminate the need of trustable third parties in electronic payment systems. Modern blockchains incorporate smart contracts [27,28] (contracts hereon), which are stateful programs stored in the blockchain that govern the functionality of blockchain transactions. Users interact with blockchains by invoking contracts[3], whose execution controls the exchange of cryptocurrency. Contracts allow sophisticated functionality, enabling many applications in decentralized finances (DeFi), decentralized governance, Web3, etc.

* This work was funded in part by PRODIGY Project (TED2021-132464B-I00)—funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR—by DECO Project (PID2022-138072OB-I00)—funded by MCIN/AEI/10.13039/501100011033 and by the ESF+—and by a research grant from Nomadic Labs and the Tezos Foundation.

[3] Non-contract addresses can be considered as unit contracts.

Contracts are written in high-level programming languages, like Solidity [2] and Ligo [4], which are then typically compiled into low-level bytecode languages like EVM [28] or Michelson [1]. Even though contracts are typically small compared to conventional software, writing contracts is notoriously difficult. The open nature of the invocation system—where every contract can invoke every other contract—facilitates that malicious users break programmer's assumptions and steal user tokens (e.g. [23]). Once installed, contract code is immutable[4], and the effect of running a contract cannot be reverted (the contract *is* the law).

Two classic reliability approaches can be applied to contracts:

- **static techniques** ranging from static analysis [26] and model checking [22] to deductive software verification techniques [3,21,8,14], theorem proving assistants [7,5,24] or assisted formal construction of programs [25].
- **dynamic verification** [15,6,18,10] dynamically inspecting the execution of contracts against specifications taking corrective measures.

We follow in this paper a dynamic monitoring technique. Monitors are a defensive mechanism to express desired properties that must hold during the execution of the contracts. If the property fails, the monitor fails the whole transaction. Otherwise, the execution finishes normally according to the contract code. In practice, monitors are mixed within the contract code, which limits the properties that can be monitored. In [10], the authors presented a hierarchy of monitors, including operation and transaction monitors. An operation monitor for a contract A runs alongside A and reads and modifies specific monitor variables stored in A [15,6,18]. Operation monitors can only execute when A is invoked and cannot inspect or invoke other contracts. Transaction monitors [10] can inspect information across a full transaction, even after the last invocation of A in the transaction. For example, the return of a loan *within the transaction* is an important property that can be monitored with a transaction monitor and not by an operation monitor, because a transaction must fail if the money lent is not returned by the end of the transaction.

Traditional blockchain systems cannot implement transaction monitors [10], but fortunately, this is easy to achieve by extending the execution model with two simple features: a `first` instruction and a Fail/NoFail hookup mechanism. Instruction `first` returns *true* during the first invocation of the contract in the current transaction. The Fail/NoFail mechanism equips each contract with a new flag, `fail`, that can be assigned (to *true* or *false*) during the execution of the contract (and that is *false* by default). The semantics of `fail` is that transactions fail if at least one contract has its `fail` flag set to *true* at the end of the transaction.

In this paper, we study an even richer notion of monitors that enables to fail or commit depending on *future transactions*. Future monitors can predicate on sequences of transactions during a bounded period of time. This period of time, called the *monitoring window* is fixed a priori.

[4] Although there are techniques to upgrade the behaviour of smart contracts, like proxy patterns and diamond proxy [19], the actual code does not change.

Optimistic rollups. Future monitors can be implemented easily in Layer-2 Optimistic Rollups[5], which are an approach to improve blockchain scalability by moving computation and data off-chain. The most popular optimistic rollup implementation is Arbitrum [9], implemented on top of the Ethereum blockchain [28]. Arbitrum offers the same API as Ethereum, allowing to install and invoke Ethereum contracts. Arbitrum transactions are executed off-chain and their effects are submitted as *assertions*. Assertions are *optimistically* assumed to be correct and a fraud-prove arbitration scheme allows to detect invalid assertions. Assertions are pending during a challenging period[6] to allow observers to check their correctness. The arbitration game consists of a bisection protocol, played between the challenger and asserter, which has the property that the honest player can always win the dispute. Assertions that survive until the end of the challenge period become permanent. Future monitors can exploit the delay imposed by the challenging period to fail or commit based on information from the future.

Bounded Future Monitoring. In this article, we enrich transaction monitors with a controlled ability to predicate about the future evolution of blockchains. Contracts are extended to include: `txid`, failmap, and timeout. The instruction `txid` returns the (unique) current transaction identifier. Each contract is equipped with a map failmap indicating—for each transaction involving the contract—whether the future monitor of the transaction is activated or not, and if so, its monitoring status (commit, fail or undecided). By default, future monitoring is deactivated. Contracts can modify their failmap (1) to activate the future monitor of the current transaction, or (2) to commit or fail undecided future monitors of previous transactions within the monitoring window. If a contract sets a past transaction failmap entry to fail, the corresponding transaction fails. The timeout function is invoked at the end of the monitoring window to decide whether to fail or commit if the future monitor of the transaction is still undecided. This guarantees that transactions cannot be pending after a bounded amount of time.

We call our monitors *future monitors* since the decision to commit or fail may depend on transactions that will execute in the future. Future monitors expand the monitor hierarchy presented in [10], which included operation and transaction monitors as well as monitors that involve several contracts (multicontract monitors) or even the whole blockchain (global monitors), but always in the context of a single transaction. When combined with future monitors, we obtain *multicontract future monitors* and *global future monitors*, but we leave these extensions as future work. A particular subclass of *multicontract future monitors* was studied in [16] focusing on long-lived transactions [17], whose lifetime span blockchain transactions and potentially involve different contracts and parties. Fig. 1 shows the updated monitoring hierarchy including future monitors.

[5] Optimistic Rollups for short.
[6] Currently a week.

2 Model of Computation

We introduce now our abstract model of computation to reason about blockchains.

Blockchains Execution Overview. Blockchains are incremental permanent records of executed transactions packed in blocks. 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, which is identified by its unique address. The execution of an operation follows the instructions of the program (the contract) stored at the destination address. Contracts can modify their local storage and invoke other contracts.

Transaction execution consists of executing operations, computing their effects (which may include the generation of new operations) until either (1) there are no more pending operations, or (2) an operation fails or the available gas is exhausted. In the former case, the transaction commits and all changes are made permanent. In the latter case, the transaction fails and no effect takes place in the storage of contracts, except that some gas is consumed. Therefore, the state of contracts is determined by the effects of committing transactions.

Model of Computation. Our model computation describes blockchain state evolution as the result of sequential transaction executions. Blockchain configurations are records containing all information required to compute transactions, such as: a partial map between addresses and their storage and balance, plus additional information about the blockchain such as block number. We use Σ to denote blockchain configurations and $\mathcal{U}$ to denote balances of external users.

Transactions are the result of executing a sequence of operations starting from an external operation placed by a user. Transactions can either commit, if every operation is successful, or fail, if one of its operations fails or the gas is exhausted. We use function $\mathsf{basicTx}$, which takes a transaction, a blockchain configuration, and balances of external users as inputs, and returns the blockchain configuration and the external user balances that result from executing the transaction in the input configuration. Additionally, predicate succ indicates whether the execution of a transaction commits or fails in a given blockchain configuration and external user balances. Furthermore, function $\mathsf{discount}$ deducts the specified amount of tokens from the balance of the indicated user in the provided external user balances. The following relation $\rightsquigarrow_{tx}$ defines the evolution of the blockchain

Present	**Future**
Global monitors	Global future monitors [future work]
Multicontract monitors	Multicontract future monitors [16] [future work]
Transaction monitors [10]	Future monitors [this work]
Operation monitors [6,15,18]	

Fig. 1. Monitor hierarchy. The first column belongs to [10].

using $\mathsf{basicTx}$, succ and $\mathsf{discount}$:

$$\texttt{commit}\,\frac{\mathsf{basicTx}(tx,\Sigma,\mathcal{U}) = (\Sigma',\mathcal{U}') \qquad \mathsf{succ}(tx,\Sigma,\mathcal{U}) = \text{commit}}{\Sigma,\mathcal{U} \rightsquigarrow_{tx} \Sigma',\mathcal{U}'} \qquad \texttt{fail}\,\frac{\mathcal{U}' = \mathsf{discount}(\mathcal{U},\mathsf{src}(tx),\mathsf{cost}(tx)) \qquad \mathsf{succ}(tx,\Sigma,\mathcal{U}) = \text{fail}}{\Sigma,\mathcal{U} \rightsquigarrow_{tx} \Sigma,\mathcal{U}'}$$

If a transaction fails (rule `fail`), the blockchain configuration is preserved, but the external user originating the transaction pays for the resources consumed. Cost and resource analysis are out of the scope of this paper, so we ignore the computation of $\mathcal{U}$.

Operation and transaction monitors are defined at the operation and transaction level, and thus, they are implemented inside $\mathsf{basicTx}$ and abstracted away in this model.

3 Bounded Future Monitored Blockchains

In this section, we present a modified model of computation supporting future monitors. The main addition is the implementation of monitoring transactions predicating on future transactions within a monitoring window k. The monitoring window captures for how long (in the number of transactions) the monitor can predicate on. This additional feature enables us to install a monitor per transaction. Future instances of contracts that activated a future monitor can decide to either fail or commit the past transaction within the monitoring window. If any contract sets to fail the transaction future monitor of a past transaction, the monitored transaction fails. Otherwise, when all contracts that monitor a given transaction commit the transaction becomes permanently committed.

3.1 Future *k*-bounded Monitors

Transactions can commit or fail depending on their subsequent k transactions, and thus, the post-state after executing a transaction may depend on future transactions. At any given point in time, transaction future monitors may:

- fail because at least one contract involved set the monitor to fail;
- commit because all contracts involved set the monitor to commit;
- stay pending.

Therefore, we identify three transaction monitor states: known to fail, (denoted by Fail), known to commit (denoted by Commit) and undecided (denoted by $\mathsf{?}$). Finally, we add another value to represent transactions without monitors: None.

Failing Map. A contract C can only interact with the future monitor of transaction t if C was involved in t. To keep track of different monitors for C (for different transactions), every contract C has a map, called *failing map*, from transactions to monitor states.

At the start of a transaction, the monitor is deactivated and can only be activated during the current transaction. Therefore, if at the end of a transaction

t no contract updated the failing map of its monitor for t, then the behavior is like legacy unmonitored transactions (as previously described in Section 2).

A contract C can modify its failing map many times but only the entries of those transactions where C was involved and activated the monitor. Changes to failing maps at the end of transactions can be (1) the activation of the monitor for the current transaction (from None to Fail, Commit, or ?, indicated by dashed arrows in Fig. 2); or (2) decisions reached for undecided monitors (from ? to Fail or Commit, indicated by plain arrows).

$$\begin{tikzcd}
\mathsf{Fail} & & \mathsf{Commit} \\
& ? \arrow[ul] \arrow[ur] & \\
& \mathsf{None} \arrow[u, dashed] \arrow[uul, dashed, bend left] \arrow[uur, dashed, bend right] &
\end{tikzcd}$$

Fig. 2. Monitor transitions.

Timeout. Contracts have a new special function called timeout that can be used to describe the decision of undecided monitors at the monitoring window. Function timeout takes a transaction identifier and returns either Fail or Commit and it is set by contracts. The default timeout function returns Commit.

At the end of the monitor window, the system invokes timeout if the failing map entry for that transaction is marked as ?. If at least one contract involved in the transaction decides to fail, the transaction fails, and otherwise the transaction commits.

3.2 Extending the Model of Computation

We extend blockchain configurations with a *future monitor context* Δ associating contracts with their failing map and timeout function.

Transaction Execution: Transactions can immediately commit or fail, or depend on future transactions that happen within the monitoring window, so the execution of a transaction can return one of the following cases:

– a new configuration as an immediate commit,
– a new configuration as an immediate fail,
– two *possible* new configurations, one for failing and one for committing, which depends on the future.

These behaviors are captured by a new function applyTx that checks if future monitors were activated during the transaction. Future monitors restrict the behavior of the blockchain, because they only modify the blockchain evolution making transactions fail more often.

Non-monitored transactions either immediately commit or fail based on function succ, and their effects are equivalent to the traditional model.

The function applyTx, when applied to a monitored not failing transaction, returns two blockchain configurations, describing the only two possible futures. The first configuration represents the effects if the transaction commits, and the second represents a failing transaction, so in these cases the post-configurations are identical to the previous configurations (modulo resources consumed).

A contract C can only modify its failing map to activate the future monitor of the current transaction or to decide future monitors that C had previously activated but not yet decided. If a contract incorrectly updates its failing map, the current transaction fails. When transactions fail, the system does not modify any failmap map or timeout function.

Blockchain System. There are two types of transactions: *permanent* (committed or failed) and *pending* transactions. *Blockchain runs* are pairs (H, τ) consisting of a sequence H of consolidated blockchain configurations called the *history* and a directed tree τ where each internal node has one or two children. H contains only permanent transaction. Tree τ is called the *monitoring tree* and includes pending transactions. Each node in the monitoring tree is a blockchain configuration. The monitoring tree represents all possible sequences of blockchain states that the list of pending transactions can generate. Exactly one path in the tree will eventually survive and become part of H, which depends on whether the corresponding transactions commit of fail. Each level in the tree corresponds to the execution of transactions up to that level but different configuration at the same level is a different possible reality. To simplify notation, we use n to refer to the blockchain configuration captured by node n in the tree. The root of the monitoring tree is the last blockchain configuration that was consolidated, that is, the last blockchain configuration in the history sequence.

The height of the monitoring tree is at most k. It can be shorter than k at the genesis of the blockchain but once the first k transactions have been executed the monitoring tree reaches and maintains a height k. In the worst case, depending on the contracts deployed in the blockchain, the monitoring tree can have $2^{k+1} - 1$ nodes, but in general not every transaction is going to be monitored which reduces the branching and hence the size of the tree.

Fig. 3 shows a blockchain run (H, τ). The first $j + 1$ transactions are permanent and the last k transactions are pending. The last permanent blockchain configuration is $(\varSigma, \varDelta)$ and it is also the root of the monitoring tree τ. When the first pending transaction, t_{j+1}, executes from configuration $(\varSigma, \varDelta)$, a contract C that executed in t_{j+1} activated the transaction future monitor generating a branching in τ. However, not all transactions generate a branching in the monitoring tree as not all transactions are necessarily monitored, (for example t_{j+k}). Configuration $(\varSigma', \varDelta')$ is a one of the possible outcomes of executing all pending operations.

Notation. We use the following functions:

- $\mathsf{nextTx}(n)$: returns the transaction that labels the outgoing edges from n.
- The *successor* of a node $n \xrightarrow{t} n'$ in the monitoring tree.
- $\mathsf{successors}(n)$: given a node n that is not a leaf, returns all successors of n, which can be (n_c, n_f), where n_c is the committing successor and n_f the failing successor, or n' if n is not branching.
- the *committing subtree* of n: the maximal subtree rooted at the committing successor of n.
- the *failing subtree* of n: the maximal subtree rooted at the failing successor of n.
- $\mathsf{allFutures}(n)$: the set of leaves reachable from node n.

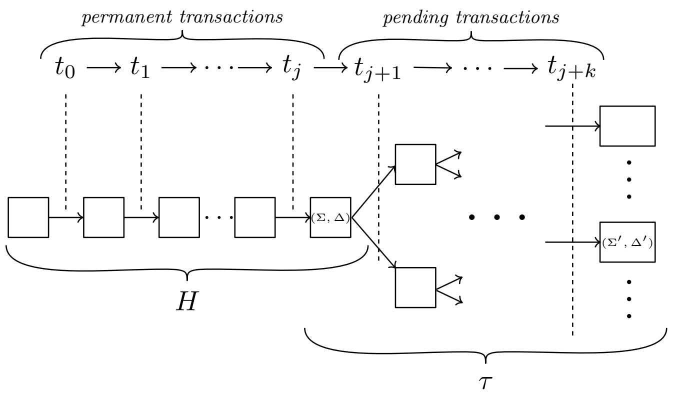
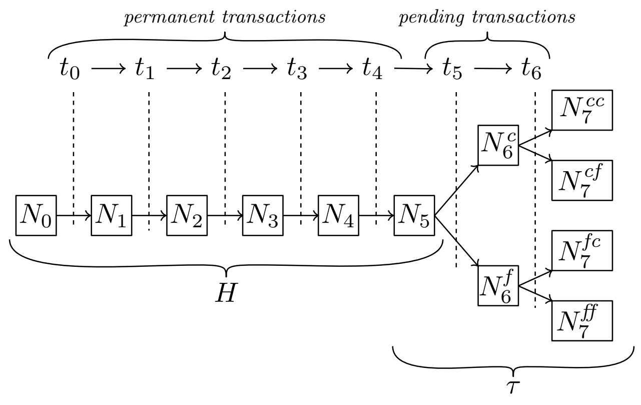

permanent transactions
pending transactions
$t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_j \longrightarrow t_{j+1} \longrightarrow \cdots \longrightarrow t_{j+k}$
(Σ, Δ)
(Σ', Δ')
H
τ

Fig. 3. A blockchain run of $j + 1$ permanent transactions and k pending transactions.

Consider $n \xrightarrow{t} n'$. The configuration at n' is one of the possible results of executing transaction t from the blockchain configuration at n. For simplicity, when referring to a monitoring tree τ with the root node n, we use the terms τ and n interchangeably. Thus, $\mathsf{successors}(\tau)$ denotes the successors of the root node of τ. The possible futures of the root node of monitoring tree τ, denoted by $\mathsf{allFutures}(\tau)$, is referred as the futures in τ.

Example 1. The following figure shows an example run after 7 transactions, starting at initial blockchain configuration N_0 and monitoring window $k = 2$.



History H corresponds to the first 5 permanent transactions. The remaining transactions are pending forming a directed tree τ whose root is N_5. The transaction at node N_5 is $\mathsf{nextTx}(N_5) = t_5$. Node N_5 successors are $\mathsf{successors}(N_5) = (N_6^c, N_6^f)$. The committing subtree of N_5 is the subtree with root N_6^c and the failing subtree of N_5 is the subtree with root N_6^f. Finally, the futures in τ are $\mathsf{allFutures}(\tau) = \{N_7^{cc}, N_7^{cf}, N_7^{fc}, N_7^{ff}\}$. We annotate with superscript c and f the committing and failing transactions, respectively, and group them in sequences describing paths in monitoring trees.

```
function step((H, τ), t)
    $\tau' \leftarrow$ attach$(\tau, t)$
    if height$(\tau') \leq k$ then
        return$(H, \tau')$
    else
        $\tau'' \leftarrow$ decide$(\tau')$
        tx $\leftarrow$ nextTx$(\tau)$
        $H.add(\tau \xrightarrow{\text{tx}} \tau'')$
        return$(H, \tau'')$
```

```
function attach(τ, t)        ▷ Extends monitoring trees.
    $\tau' \leftarrow \tau$
    for $l \in$ allFutures$(\tau)$ do
        switch applyTx$(t, l)$ do
            case Commit$(l_c)$ : $\tau'$.add$(l \xrightarrow{t} l_c)$
            case Fail$(l_f)$ : $\tau'$.add$(l \xrightarrow{t} l_f)$
            case Pending$(l_c, l_f)$: $\tau'$.add$(l \xrightarrow{t} (l_c, l_f))$
    return $\tau'$
```

Fig. 4. Functions step and attach.

3.3 Blockchain evolution

The evolution of the blockchain is defined by function step (see Fig. 4) which takes blockchain runs and transactions, and extends runs. The system has only one rule:

$$\frac{\mathsf{step}((H,\tau),t) = (H',\tau')}{(H,\tau) \twoheadrightarrow_t (H',\tau')}$$

Valid traces are defined by the relation $\twoheadrightarrow$ and consist of chains of related blockchain states $(H_0, \tau_0) \twoheadrightarrow_{t_0} (H_1, \tau_1) \twoheadrightarrow_{t_1} \dots$ where (H_0, τ_0) is an initial blockchain run with $\tau_0 = H_0 = (\Sigma, \Delta)$.

Let (H, τ) be a blockchain run and t a transaction. We extend the monitoring tree τ by adding a new level attaching t from every possible leaf, which increases by one the height of τ (see Fig. 4). Let τ' be the result of attach(τ, t). If τ' has height $k + 1$, the monitoring window for the first transaction in τ' has expired and its monitor must fail or commit. To take this decision, function step invokes function decide. The resulting monitoring tree τ'' returned by function decide becomes the new monitoring tree. Finally, function step extends H making the first pending transaction permanent.

Function decide (see Fig. 5) determines whether to commit or fail the first pending transaction tx in monitoring tree τ with height $k + 1$ returning either the committing or failing subtree of τ. If τ has only one successor, the decision is trivial, otherwise we analyze tx possible futures. Function decide checks all futures assuming tx commits, (i.e., all leaves in the committing subtree of τ); if the future monitor of transaction tx commits in all of them, then tx commits and the committing subtree of τ becomes the new monitoring tree. Otherwise, tx fails and the failing subtree of τ becomes the new monitoring tree. If decide cannot assert whether the monitored transaction fails or commits, decide invokes timeout to decide (see function knownToCommitWithTimeout in Fig. 5).

In some cases, the decision of future monitors is known before the monitoring windows ends. In such instances, some nodes are unreachable, called *impossible nodes*. For example, when a transaction future monitor is waiting for a transaction in the future and that transaction happens before the monitoring window ends, the future monitor is going to be set to commit, which turns all nodes

```
function decide($\tau$)                    ▷ Decides commit/fail of the root transaction of $\tau$
    assert height($\tau$) = $k + 1$
    $\tau' \leftarrow$ prune($\tau$)
    $t \leftarrow$ nextTx($\tau$)
    switch successors($\tau'$) do
        case $\tau''$: return $\tau''$
        case $(\tau_c, \tau_f)$:
            if $\forall l \in$ allFutures($\tau_c$) : knownToCommitWithTimeout($l, t$) then return $\tau_c$
            else return $\tau_f$
```

```
function prune($\tau$)
    if $\tau$ is a leaf then return $\tau$
    $t \leftarrow$ nextTx($\tau$)
    switch successors($\tau$) do
        case $\tau'$: return $\tau \xrightarrow{t}$ prune($\tau'$)
        case $(\tau_c, \tau_f)$:
            $\tau'_c \leftarrow$ prune($\tau_c$)
            $\tau'_f \leftarrow$ prune($\tau_f$)
            if $\forall l \in$ allFutures($\tau'_c$) : knownToCommit($l, t$) then return $\tau \xrightarrow{t} \tau'_c$
            if $\forall l \in$ allFutures($\tau'_c$) : knownToFail($l, t$) then return $\tau \xrightarrow{t} \tau'_f$
            return $\tau \xrightarrow{t} (\tau'_c, \tau'_f)$
```

```
function failmapCommit($\Delta, c, t$) return $\Delta[c]$.failmap[$t$] = Commit
function failmapFail($\Delta, c, t$) return $\Delta[c]$.failmap[$t$] = Fail
function timeoutCommit($\Delta, c, t$) return $\Delta[c]$.timeout[$t$] = Commit
function undecided($\Delta, c, t$) return $\Delta[c]$.failmap[$t$] = ?
function monitoringContracts ($l, t$) return $\{c : l.\Delta[c]$.failmap[$t$] $\neq$ None$\}$
function knownToCommit ($l, t$)
    return $\forall c \in$ monitoringContracts($l, t$) : failmapCommit($l.\Delta, c, t$)
function knownToFail ($l, t$)
    return $\exists c \in$ monitoringContracts($l, t$) : failmapFail($l.\Delta, c, t$)
function commitWithTimeout($\Delta, c, t$)
    return failmapCommit($\Delta, c, t$) $\vee$ (undecided($\Delta, c, t$) $\wedge$ timeoutCommit($\Delta, c, t$))
function knownToCommitWithTimeout ($l, t$)
    return $\forall c \in$ monitoringContracts($l, t$) : commitWithTimeout($l.\Delta, c, t$)
```

Fig. 5. Functions decide, prune and auxiliary functions.

in its failing subtree impossible nodes. Concretely, if in all possible futures in the committing subtree of node n its transaction is known to commit, then all nodes in the failing subtree of n are impossible nodes. Similarly, if in all possible futures in the committing subtree of node n its transaction is known to fail, then all nodes in the committing subtree of n are impossible. Impossible nodes are removed before deciding whether a transaction commits or not, since we may incorrectly deduce that a monitor fails because of an impossible future

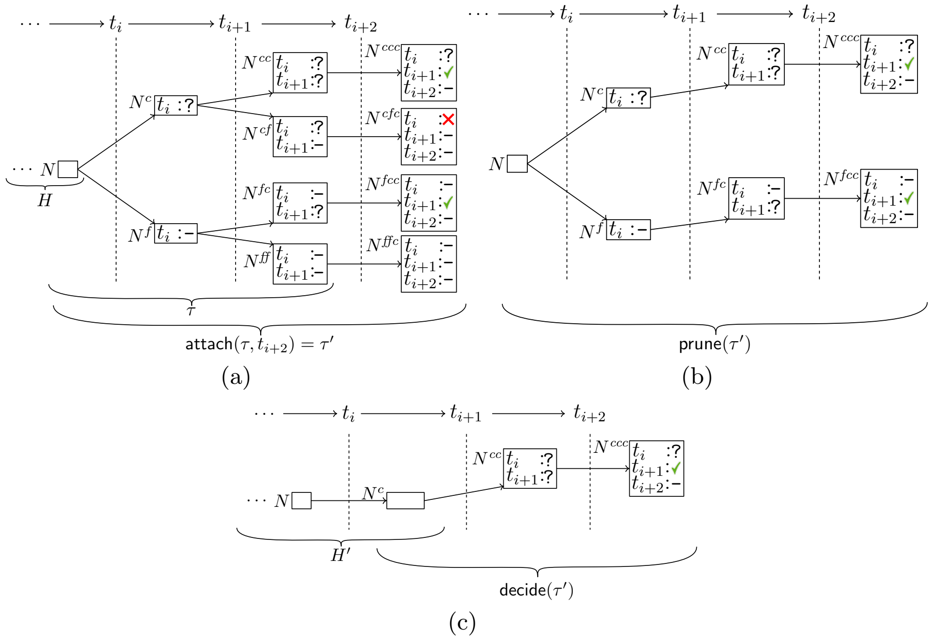


Fig. 6. Application of function step in a blockchain run.

node. Consequently, decide invokes prune to remove all impossible nodes, and only then, decide determines whether the root transaction commits or not as explained above.

Function prune (see Fig. 5) shows how to prune impossible nodes from trees. To guarantee that impossible nodes are pruned before checking if roots of trees are impossible (either commit or fail), we perform a bottom-up recursion.

Example 2. Fig. 6 shows the result of applying function step to blockchain run (H, τ) with a monitoring window $k = 2$ and two pending transactions t_i and t_{i+1}. Each node in the monitoring tree is annotated with the monitor state of all pending transactions up to that node: a question mark means undecided monitors, a tick means known to commit monitors, a cross means known to fail monitors, and a dash denotes no monitored transactions. Initially, no monitors are decided in any node in τ.

Function step$((H, \tau), t_{i+2})$ first invokes function attach(τ, t_{i+2}). This function adds a new level to τ by applying transaction t_{i+2} at all leaves in τ, obtaining monitoring tree τ', Fig. 6(a). Transaction t_{i+2} immediately commits at all leaves in τ, generating nodes N^{ccc}, N^{cfc}, N^{fcc} and N^{ffc}. The future monitor for transaction t_i is known to fail at node N^{cfc} while remaining undecided at node N^{ccc} and the future monitor for transaction t_{i+1} is known to commit at nodes N^{ccc} and N^{fcc}. Next, as the height of the new monitoring tree, τ', is $3 > 2$, function step invokes function decide(τ') to decide if the first pending transaction, t_i, fails or commits. Function decide invokes function prune to remove all impossible nodes in τ'. When computing prune, the failing subtree of node N^c, rooted at node N^{cf}, is removed because at node N^{ccc} the future monitor for the transaction at node N^c, t_{i+1}, is known to commit and node N^{ccc} is the only future in the committing subtree of node N^c, making the subtree rooted at N^{cf} an impossible subtree. Similarly, the subtree rooted at N^{ff} is an impossible subtree and it is also removed by function prune.

Subtrees with roots N^{cf} and N^{ff} are the only ones removed when applying function prune to monitoring tree τ', as shown in Fig. 6(b).

Finally, to decide whether to commit or not transaction t_i function decide consider node N^{ccc}, as it is the only future in the committing subtree of node N in the monitoring tree returned by function prune. At node N^{ccc} the future monitor for transaction t_i is undecided. However, since its monitoring window has ended, function decide uses the timeout of the contracts that are undecided. Assuming for all undecided contracts their timeout function commit transaction t_i, then function decide commits transaction t_i, returning the subtree rooted at N^c as the new monitoring tree (see Fig. 6(c)), it would fail if at least one contract timeout function fails. Finally, function step extends H by making transaction t_i permanent. If prune had not been applied before function decide evaluated all futures in the committing subtree of N, transaction t_i would have incorrectly failed, as in impossible future N^{cfc}, the future monitor for transaction t_i fails.

An example of contracts that only lend their tokens if they receive them back within 2 transactions in the future can be found in [11].

4 Properties

We discuss now properties of the model of computation defined in Section 3. In particular, we establish how the new model extends the previous one, that the size of monitoring trees is manageable, and the blockchain always progresses. We assume a fixed monitoring window k. All proofs can be found in [11].

After the monitoring window has expired, the root transaction is confirmed and one of two possible successors is consolidated.

Lemma 1. *Let (H, τ) be the system run after k transactions, t a transaction and $(H', \tau') = \mathsf{step}((H, \tau), t)$. The root of τ' is one of the successors of the root of τ and all paths in τ' without leaves are also paths in τ. Moreover, H' is obtained by extending H with the first pending transaction on τ.*

The first k transactions from the genesis are just added to the tree. From the previous lemma, after k transactions and when a new step is taken, the first pending transaction is either committed or failed and a new pending transaction is attached to all leaves. Moreover, the transaction added to the history is the root of the previous monitoring tree and one of its successors is the root of the new monitoring tree. In other words, exactly one of the paths in the monitoring tree eventually becomes permanent, and thus, the blockchain always progresses.

Corollary 1 (Progress). *Function* step *is total and, after the first k invocations, each execution of* step *makes one transaction permanent.*

The height of the monitoring tree is bounded by the monitoring window.

Lemma 2 (Bounded Certainty). *Let τ be a monitoring tree in a blockchain run obtained by applying function step l times. Then, the height of τ is the minimum between l and k. Moreover, all leaves in τ are in its last level.*

Function prune removes all impossible nodes from monitoring trees. Function prune recursively removes impossible nodes in the committing and failing subtrees, and then, determines if it can remove any subtree by inspecting all possible futures in the committing successor.

Lemma 3. *Function* prune(τ) *returns a sub-monitoring tree of τ without impossible nodes and only impossible nodes were removed.*

Function step consistently makes the blockchain progress. After more than k transactions were added, the first pending transaction is made permanent (see Corollary 1). The resulting monitoring tree keeps the order of the rest of the pending transactions and it also preserves the same information of the pending transactions except the last.

Lemma 4. *Let τ be a monitoring tree, η be the result of expanding τ with a new transaction, t be the first pending transaction in τ, and ν be the decided subtree of η.*

- *If η has only one successor then ν is the result of pruning η's successor.*
- *If η has two successors, then let η_c and η_f be the result of pruning the committing and failing subtrees of η respectively.*
  - *Monitoring tree ν is η_c if in all possible futures assuming t commits, transaction t does not fail or if no decision has been reached, all pending* timeout *functions of t commit.*
  - *Monitoring tree ν is η_f if there is a possible future where assuming transactions t commits, leads to the monitor of t fail or some of the pending* timeout *function of t fail.*

The size of monitoring trees can be exponential in the number of monitored transaction rather than in the monitoring window size, as monitored transactions are the only ones branching monitoring trees.

Lemma 5. *Let τ be a monitoring tree and m be the number of monitored transactions in τ (so $m \leq k$). Then, the size of τ is in $\mathcal{O}(2^m \times k)$.*

In practical scenarios, the number of monitored transactions typically is small compared to the monitoring window because most transactions do not require future monitors. This makes the size of the monitoring tree much smaller than the theoretical maximum.

Corollary 2. *If the number of monitored transactions in monitoring trees is constant then the size of monitoring trees is bounded by $\mathcal{O}(k)$.*

Finally, we show that adding future bounded monitors preserves legacy executions, so for blockchain runs where no contracts use future monitors, the monitoring tree is a chain with no branching.

A legacy monitoring tree τ is such that every configuration obtained from applying $\mathsf{applyTx}$ coincides with rule $\rightsquigarrow$.

Lemma 6 (Legacy Pending Transactions). *Let τ be a legacy monitoring tree. Then, τ is a chain and the effect of executing all transactions in τ is equivalent to executing them in the traditional model of computation.*

If we add that the permanent history is equivalent (up to now) to the traditional model, then the evolution of the blockchain in both models coincide.

Lemma 7 (Legacy History). *Let τ be a legacy monitoring tree and H be a history such that every permanent transaction coincides with rule $\rightsquigarrow$. Then, the result of concatenating H and τ is equivalent to the traditional model of computation.*

From Corollary 1 and Lemma 7, we conclude that the new model of computation is consistent with the previous model of computation and eventually creates a chain. Additionally, Corollary 2 implies that in practical scenarios, the size of monitoring trees is linear on the monitoring window, making it a feasible and practical blockchain implementation.

5 Atomic Loans

Flash loan contracts allow other contracts to borrow tokens *without any collateral* only if the borrowed tokens are repaid during the same transaction [12] (typically with some interest). Atomic loans are a generalization of flash loans where the borrowing party can repay the lending party in future transactions. It is not possible to implement flash loans unless additional mechanisms are added to the blockchain [10]. Similarly, it is impossible to implement atomic loans in traditional blockchain computational models. As transaction monitors [10] enable flash loans transactions, future monitors allow monitors to check properties across transactions enabling atomic loans. We illustrate now how to implement atomic loans using the monitoring window as the maximum payback time.

We specify lender contracts as contracts respecting the following two properties:

Specification 1 (Atomic Loans) *We say contract* A *is an atomic lender if:*
AL-safety: *A loan from* A *is repaid to* A *within the monitoring window.*
AL-progress: *Contract* A *grants loans unless* **AL-safety** *is violated.*

The following contract `FlashLoanLender` shows a simple contract implementing a flash loan lender[7] using Fail/NoFail hookup [10], i.e. with no future monitors but transaction monitors. We highlight monitor code with gray background.

[7] Flash loan lender are atomic loan lenders with paying back window of one.

```
contract FlashLoanLender {
  uint pending_returns = 0;
  uint fee;
  function lend(address payable dest, uint amount) public
    { require(amount <= this.balance);
      dest.receiveLoan{value: amount}(fee);
      pending_returns += amount + fee;
      this.fail = (pending_returns != 0); }
  function returnLoan() external payable
    { pending_returns -= msg.value;
      this.fail = (pending_returns != 0); } }
```

Function `lend` lends as long as the lender has enough funds, annotates the borrowed tokens in `pending_returns` and sets its `fail` bit so the transaction commits only if the loan is paid back. When the loan is returned, `returnLoan` decreases `pending_returns` and updates its `fail` bit. At the end of each transaction, if there are pending loans the `fail` bit will make the transaction fail.

The above contract implements flash loans that must be returned within a transaction, but does not work properly if future transactions are considered. It is not possible to successfully predict or check whether the loan is returned in some future transactions. We show now how future monitors solve this problem.

The following contract `Lender` is an atomic lender using future monitors. All loans are treated equally and should be paid back on time, and if one loan is not returned, then all loans issued at the same transaction would be rejected. Here we are being too strict compared to practical cases, but it is enough to illustrate the use of future transaction monitors.

```
contract Lender {
  uint fee;
  function lend(address payable dest, uint amount) public
    { require(amount <= this.balance);
      dest.receiveLoan{value: amount}(fee);
      pending_returns[msg.txid] += amount + fee;
      if(pending_returns[msg.txid] != 0)
         this.failmap[msg.txid] = UNDECIDED; }
  function returnLoan(txId id) public
    { pending_returns[id]-= msg.value;
      if(pending_returns[id] == 0) this.failmap[id] = COMMIT; }
  } with monitor {
    map<txId, int> pending_returns;
    function timeout(txId id) { return FAIL; } }
```

Contract `Lender` uses a map `pending_returns`, from transactions to the amount borrowed within that transaction, to determine whether a transaction should commit or fail. Function `lend` grants a loan if the lender has enough funds, increases the corresponding entry in map `pending_returns` for the current transaction and sets the `failmap` entry activating the current transaction monitor. Client contracts can repay loans by invoking `returnLoan`, which receives the transaction identifier of the lending transaction to decrease the corresponding

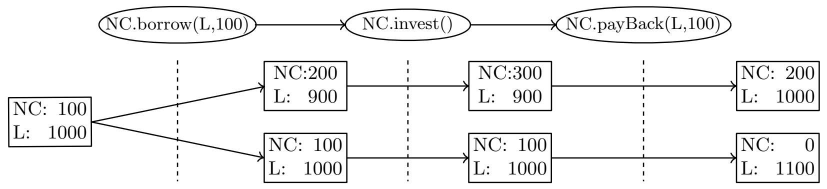


Fig. 7. Balance of contracts `NC` and `L` in the monitoring tree after executing the three transactions posted by a client.

entry in `pending_returns` by the amount received. If `pending_returns` reaches 0 for a given transaction, the `failmap` entry of that transaction is set to `COMMIT`. Finally, `timeout` returns `FAIL` to fail transactions with unpaid loans at the end of their monitoring window.

Clients can request loans without further collateral, satisfying **AL-progress**, and if loans are not returned within the monitoring window, the lending transaction will retroactively fail, satisfying **AL-safety**.

The following contract `NaiveClient` requests a loan invoking `borrow`.

```
contract NaiveClient {
  map<pair<txId,Lender>, uint> toPay;
  function borrow(Lender l, uint amount) onlyOwner
    { l.lend(amount);
      toPay[(msg.txid(),l)] = amount; }
  function receiveLoan(uint fee)
    { toPay[(msg.txid,msg.sender())] += fee; }
  function invest() onlyOwner { ... }
  function payBack(Lender l, uint amount, txId id) onlyOwner
    { require(toPay[(id,l)] >= amount);
      toPay[(id,l)] -= amount;
      l.returnLoan{value: amount}(id); } }
```

In subsequent transactions, the client can invest the funds, and in a final transaction, return the loan to the lender invoking `payBack`.

Let `NC` and `L` be two contracts installed in a blockchain with a monitoring window of length 2, where `NC` runs `NaiveClient` and `L` runs `Lender`. Consider (Σ, Δ) to be the current state of the blockchain at which `NC` has 100 tokens and `L` has 1000 tokens. From (Σ, Δ), the sequence of transactions is: (1) `NC` requests a loan, (2) `NC` invests assuming contract `L` lends the money, and (3) `NC` returns the loan. Because `L` employs future monitors to guarantee clients pay back, the first transaction generates a branching on the blockchain evolution. The next two transactions are not monitored, thus they do not create any branching. Therefore, after these three transactions, there exist two possible futures as shown in Fig.7, one where `L` grants the loan and another where it does not. We can see that `NC` pays back in all possible futures. Moreover, contract `NC` pays back even in the future where contract L fails the past lending operation (for a detailed explanation see [11]).

A malicious lender can take advantage of such behavior, for example using the following contract MaliciousLender.

```
contract MaliciousLender {
  uint fee;
  function lend(address payable dest, uint amount) public
    { dest.receiveLoan{value: amount}(fee);
      this.failmap[msg.txid] = UNDECIDED; }
  function returnLoan(txId id) public { return; }
  } with monitor {
    function timeout(txId id) { return FAIL; } }
```

The above malicious lender, upon receiving a loan request in function lend, if it has enough tokens, it grants the loan and marks the transaction as undecided using its failmap map. However, this lender contract does not update its failmap map when receiving paybacks. Therefore, at the end of the monitoring window, the monitor remains undecided making the lending transaction fail due to the timeout function. In other words, the malicious lender never lends any tokens, as all its loans are reverted, but it looks like it does. When combined with NaiveClient and the same three transactions described earlier, the malicious lender will receive the repayment of a loan from client NC without having given the loan. In Fig. 7, the bottom branch is the one that survives when the lender implements a malicious contract.

The problem arises because client NC does not implement any mechanism to check in which branch it is executing when repaying the loan. The naive contract does not distinguish between the scenario where the loan will ultimately be committed and the scenario where it will fail. As a result, client NC ends up providing payments in both cases.

The following contract Client presents a correct client implementing two maps,requested and toPay, to keep track of the amounts requested from lenders and its debts owed to lenders, respectively.

```
contract Client {
  map<pair<txId,Lender>, uint> toPay, requested;
  function borrow(Lender l, uint amount) onlyOwner
    { l.lend(amount);
      requested[(msg.txid,l)] = amount; }
  function receiveLoan (uint fee)
    { require(requested[(msg.txid,msg.sender)] == msg.value);
      requested[(msg.txid,msg.sender)] = 0;
      toPay[(msg.txid,msg.sender)] = msg.value + fee; }
  function invest() onlyOwner { ... }
  function payBack(Lender l, uint amount, txId id)
    { require(toPay[(id,l)] >= amount);
      toPay[(id,l)] -= amount;
      l.returnLoan{value: amount}(id); } }
```

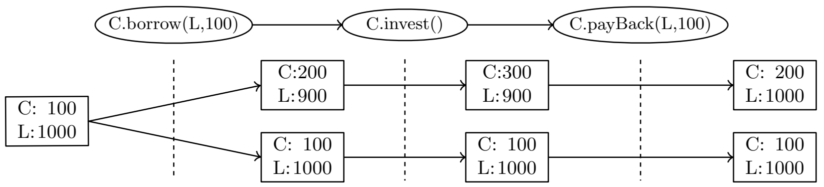


Fig. 8. Balance of contracts `C` and `L` in the monitoring tree after executing the three transactions posted by a client.

The above contract allows clients to determine the specific path in which it is executing, and thus, to decide whether to repay. Consequently, clients can successfully get loans from correct lenders while being resistant to attacks from malicious lenders.

Fig. 8 shows an execution following the same transactions as before but with the correct contract `Client`: clients request a loan, invest the money, and payback the loan. The top branch shows the case where the lender sends the money and the client returns it, while the bottom branch shows the case where the loan is not given. In the former cases, the client returns the money, and in the latter case, the client just fails the transaction.

These examples show how even contracts not monitoring transactions need to be aware that transactions can create potential executions in the blockchain evolution that may be reverted due to future monitors. Since the same transaction is executed in all possible scenarios, but their effects may be different, contracts need to know in which temporal line they are executing and act accordingly. Contract `Client` accomplishes this by maintaining a record of debts owed to lenders in variable `toPay`.

6 Related Work

Dynamic verification of smart contracts Runtime monitoring tools like ContractLarva [15,6] and Solythesis [18] take a smart contract code and its properties as input and produce a safe smart contract that fail transactions violating the given properties. They achieve this be injecting the monitor into the smart contract as additional instructions. Therefore, these monitors are restricted to one operation in a single contract. Transaction Monitors [10] extend monitoring beyond a single operation to observe the effect of an entire transaction execution on a given contract.

While these existing works provide strong foundations for smart contract verification, none directly address the ability to react based on future transactions, as proposed in this work.

Branching Computational Models The monitoring tree generated by pending transactions might reassemble the tree-like structure in branching-time logic such as CTL [13]. However, the branching in the monitoring tree represents all possible futures given by the monitors of the pending transaction, and exactly one path eventually consolidates. In particular, future monitors are not aware of the existence of other paths in the monitoring tree and therefore cannot reason about them. CTL, on the other hand, can be used to express properties that reason about different paths in the tree.

7 Conclusion

We presented future monitors for smart contracts. Future monitors are a defense mechanism enabling contracts to state properties across multiple transactions. These kinds of properties are motivated by long-lived transactions, in particular by atomic loans, which are not implementable in their full generality in current blockchains. To implement future monitors, we introduced the notion of monitoring window and two additional new mechanisms to blockchains, namely failing maps and timeout functions.

Future monitors delay the consolidation of transactions, but the system remains consistent and we gain in expressivity. The outcome of transactions remains deterministic and depends solely on the transactions themselves, but now transactions can fail because of future actions. Combining all elements we obtained a deterministic semantics with future monitors in place.

We have also illustrated that contracts need to be aware of the existence of possible executions. Future monitors introduce a branching model to describe the evolution of blockchain systems where transactions may commit or not, caused by the temporary uncertainty regarding the effect of pending transactions. Consequently, when new transactions are added to the blockchain, they are executed in multiple blockchain configurations, representing possible timelines. Therefore, contracts need to be aware of the different contexts in which they are executing, ensuring that the transaction produces the desired effects in all possible realities.

The main contribution of this paper is theoretical and we left the full implementation of future monitors as future work. Optimistic rollup systems, where the effect of transactions is already delayed due to the fraud-prove arbitration scheme, present an ideal environment to incorporate future monitors into practical blockchain systems without further implications. In particular, optimistic rollup systems can allow future transaction monitors with little modifications, and more importantly, without modifying the underlying blockchain.

For simplicity, we have neglected a specific analysis of the additional gas consumption that arises for using future monitors, which might lead to the failure of accepting transactions. Nevertheless, we conjecture that future monitors are simple enough to guarantee that a calculable amount of gas will prevent gas failing situations. However, we leave a detailed study for future work.

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)*, number 12478 in 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. In *Proc. of the FM 2019 International Workshops, Part I*, volume 12232 of *LNCS*, pages 368–379. Springer, 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. L. Bousfield, R. Bousfield, C. Buckland, B. Burgess, J. Colvin, E. Felten, S. Goldfeder, D. Goldman, B. Huddleston, H. Kalonder, F. Lacs, H. Ng, A. Sanghi, T. Wilson, V. Yermakova, and T. Zidenberg. Arbitrum nitro: A second-generation optimistic rollup. `https://github.com/OffchainLabs/nitro/blob/master/docs/Nitro-whitepaper.pdf`, 2022.
10. M. Capretto, M. Ceresa, and C. Sánchez. Transaction monitoring of smart contracts. In T. Dang and V. Stolz, editors, *Proc. of the 22nd Int'l Conf. on Runtime Verification (RV'22)*, volume 13498 of *LNCS*, pages 162–180. Springer, 2022.
11. M. Capretto, M. Ceresa, and C. Sánchez. Monitoring the future of smart contracts. *arXiv*, abs/XXXX.XXXXX, 2024.
12. A. C. Cañada, F. Kobayashi, fubuloubu, and A. Williams. Eip-3156: Flash loans. `https://eips.ethereum.org/EIPS/eip-3156`.
13. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.
14. 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.
15. 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.
16. J. Ellul and G. J. Pace. Optional monitoring for long-lived transactions. In *Proc. of the 5th ACM Int'l Workshop on Verification and mOnitoring at Runtime EXecution, Virtual Event(VORTEX'21)*, pages 35–39. ACM, 2021.

17. J. Gray. *The Transaction Concept: Virtues and Limitations*, page 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
18. 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.
19. N. Mudge. Erc-2535: Diamonds, multi-facet proxy. `https://eips.ethereum.org/EIPS/eip-2535`, February 2020. Ethereum Improvement Proposals, no. 2535.
20. S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009.
21. 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.
22. A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. VerX: Safety verification of smart contracts. In *Proc of the 41st IEEE Symp. on Security and Privacy (S&P'20)*, pages 1661–1677. IEEE, 2020.
23. D. Phil. Analysis of the DAO exploit. `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`, 2016.
24. 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.
25. I. Sergey, A. Kumar, and A. Hobor. Scilla: a smart contract intermediate-level LAnguage. *CoRR*, abs/1801.00687, 2018.
26. 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 Symp. on Security and Privacy (S&P'21)*. IEEE, May 2021.
27. N. Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 16, 1996.
28. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.