Offchain Runtime Verification (for The Tezos Blockchain)*

Margarita Capretto[1,2], Martin Ceresa[1], Felipe Gorostiaga[1,3], Fernando Macias[1], Paloma Pedregal[1], and Cesar Sanchez[1]

[1] IMDEA Software Institute, Pozuelo de Alarcón s/n, 28223 Madrid, Madrid, Spain
[2] UPM, Madrid, Spain
[3] CIFASIS,Argentina

Abstract. In this paper, we present *offchain runtime verification*, a dynamic analysis technique to inspect blockchain executions without affecting the blockchain itself.
Runtime verification (RV) is a technique that analyzes traces of system execution based on monitors created from system specifications. There are two flavors of RV: online and offline. In online RV, monitors run in tandem with the system, either with their own resources or as code inlined in the system implementation. In offline RV, monitors have a dump of the system trace available. Examples of offline monitoring include post-mortem analysis and log inspection.
We present a novel notion of monitors running offchain while fetching information about the blockchain evolution and its agents (e.g. external users, bakers) to assess security and fairness, assign blame, and compute explanations. Our monitoring infrastructure is both *online*—as the monitors can receive new blocks incrementally—and *offline* since the monitors can query the history of the blockchain. Online queries are necessary because monitors are created after the blockchain has been running and relevant information is discovered online (e.g. who interacted in the past with an address recently discovered to be malicious). We describe in this paper an RV infrastructure for offchain monitoring for the Tezos Blockchain.

1 Introduction

Blockchains [22] running smart contracts [32, 33] provide a trusted third party where transactions are persistent and permanent. Smart contracts are immutable pieces of code (the code is the contract) that govern the interaction between agents using a blockchain without requiring a trusted centralized authority. We can use smart contracts to describe sophisticated functionality, enabling many

* 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.

applications like decentralized finances (DeFi), decentralized governance, and Web3. Smart contracts are deterministic, i.e. the effects of executing smart contracts are uniquely determined by the blockchain state and the transactions parameters. Since smart contracts are immutable and they govern the blockchain evolution (including the cryptocurrency exchanged), the correctness of smart contracts is crucial and errors and vulnerabilities can lead to huge losses (e.g. [27]). Both static [1, 6, 7, 10, 23, 26, 31]and dynamic techniques [3, 8, 12, 21] have been proposed to approach the problem of smart contract correctness. Dynamic techniques analyze the evolution of the blockchain. Specifications describe correctness criteria for smart contracts and monitoring code is generated which extends the code of the contract. At runtime, monitors inspect smart contract invocations, detecting violations and reverting illegal executions. This *onchain* monitoring approach requires monitors to be deployed on-chain as part of smart contracts themselves, because otherwise, once smart contracts commit their effects cannot be reverted. Onchain monitoring, in turn, affects the normal execution of smart contracts as monitors consume some gas.

In this paper, we explore an alternative monitoring technique where monitors are deployed in a running system. Additionally, we seek non-intrusive monitors, so that the execution of smart contracts is completely unaffected by the execution of monitors. In these scenarios, monitors only observe a suffix of the system original trace. There are three possible approaches to cope with this lack of past observability: (1) *ignore the missing past* so monitors operate as if they were observing the whole history, which is the simplest approach but can lead to inaccurate results; (2) *encode the lack of knowledge* by modifying the specification [19]; (3) *access a log system* to fetch the missing past and then continue monitoring online with future events. We propose in this paper to follow the third approach by combining *offline* and *online* monitoring.

Runtime verification (RV) is a formal method for analyzing execution traces, one at a time. Traces are evaluated against monitors built from a given formal specification [5,20]. Formal specifications are described using different languages implementing different logics like linear temporal logic [28]. Most RV languages describe a monolithic monitor that processes input events. Another approach is dynamic parametrization, also known as parametric trace slicing, which quantifies over objects and spawns monitors that follow independently the objects observed as in Quantified Event Automata (QEA) [4]. One can think of it as grabbing a magnifying glass when required.

Our approach is based on Stream Runtime Verification (SRV), pioneered by Lola [11], which relates output streams of verdicts to input streams of observations. Originally designed for testing synchronous hardware, SRV has since extended to other applications, including asynchronous and real-time systems (e.g. [15]). HLola [9, 16, 18] is an implementation of Lola as an embedded DSL in Haskell, which simplifies the specification development and runtime system. HLola leverages Haskell data types for Lola streams. Our main technical contribution is the extension of HLola with functionalities for *retroactive dynamic parametrization*, where parametrized specifications can be specialized with infor-

mation discovered dynamically (parametrization) and specifications can revisit past events to obtain missing information (retroactivity).

The inspection of the blockchain evolution is a perfect application of retroactive dynamic parametrization. The blockchain state, that is, the state of each smart contract and wallet, is connected to the next state and monitors can observe the whole execution history of the system. The challenge is to design efficient monitoring runtime systems that compute only what is necessary to produce a verdict.

Previous works have already inspected the blockchain evolution, mostly for security concerns within blockchain ecosystems [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]In general, these works focus on specific problems and are tailored for performance but not correctness. In contrast, our framework is designed to be applied to several scenarios, and it allows writting diverse specifications ensuring that the monitors generated are correct with regard to the specification.

Running Example: Sandwich attacks. We introduce our running example, a blockchain vulnerability known as sandwich attack. Decentralized Exchanges (exchanges from now on) are a common application ofy blockchains to decentralized finances (DeFi) where users trade tokens directly without intermediaries. Sandwich attacks exploit the delay between transaction submission and transaction execution. When transactions are submitted for execution in blockchains they are first added as requests to a distributed service, known as mempools, containing transaction requests that have not yet been executed. The mempool content is visible to all agents in the blockchain, including malicious actors which inspect the mempool searching for victim transactions. In a sandwich attack, a victim tries to trade a large amount of token `A` for `B` in an exchange which will cause an increase in the price of token `B`. A malicious actor tries to "sandwich" the victim transaction with two additional transactions to profit from the future value increase of `B`. The first transaction, known as the *frontrunning* transaction, buys tokens `B` and is scheduled before the victim transaction. In the second transaction, known as the *backrunning* transaction, the malicious actor sells the purchased tokens `B` at a major price obtaining a profit.

The frontrunning transaction increases the price of token `B` causing the victim transaction to purchase `B` at a higher price than expected. Since token prices often fluctuate, the price of a given token when a transaction is submitted might differ from the actual transaction execution. Hence, most exchanges offer a way to define price ranges where token purchases can occur, which limits the amount of tokens `B` that the malicious actor can buy in the frontrunning transaction. This limits malicious actor's profits, but it does not prevent sandwich attacks from happening.

Consider a user trading a large amount of Tezos (XTZ) for USDT on an exchange. A malicious account can perform a sandwich attack purchasing USDT with XTZ in the same exchange right before the victim's transaction, and then selling USDT for XTZ right after.

In the remainder of the paper, we focus on detecting sandwich attacks against a specific account, denoted by `a`. We say that an account is *malicious* if it per-

forms a sandwich attack to account `a`. We also identify *suspicious* wallets as those that interacted directly with malicious accounts.

We implemented retroactive dynamic parametrization in HLola and report the result of applying our implementation to detect sandwich attacks in the Tezos blockchain [14]. An early prototype of this technique [25] was already used to efficiently detect distributed denial of service attacks in realistic network traffic. The contributions of this paper are:

- A new monitoring technique and its application for inspecting blockchain histories, described in Section 3.
- A demonstration of how to apply the features of our framework to detect sandwich attacks, identify involved actors, and compute attackers' profits and victims' losses, shown in Section 4.
- Further applications of our monitoring framework, presented in Section 5.

2 Preliminaries

We introduce now necessary concepts of Blockchains and SRV.

Blockchains. Blockchains [22] were introduced as distributed infrastructures that eliminate the need of trust third parties in electronic payment systems. Modern blockchains incorporate smart contracts [32, 33], stateful programs stored in blockchains controlling the functionality of blockchain transactions. Users interact with blockchains by invoking smart contracts. Blockchain "actors" (users and smart contracts) are identified by their *account*. We refer to accounts managed by end-users as *wallets*.

A *node* is a machine that stores a copy of the blockchain (or at least a portion of it) and keeps its local copy updated by regularly communicating with other nodes in a peer-to-peer network. Public blockchains allow anyone to launch a fully functional node. While nodes hold the entire history of the blockchain, searching this data directly can be slow and resource-intensive. Therefore, there is an ecosystem of tools, called *indexers*, that retrieve information from nodes and process it to allow efficient search. Indexers crawl the whole blockchain and store its data plus some additional information about the evolution of the blockchain, offering an API to query this information. Each API restricts the vision of the blockchain to what can be retrieved by such API language.

Stream Runtime Verification. Stream Runtime Verification (SRV) enriches monitoring algorithms from runtime verification to handle arbitrary data. SRV separates the logic of how data relates over time from the specific operations of each datatype. In this paper, we use the extensible tool HLola [9, 16, 18], an implementation of Lola [11] developed as an embedded DSL in Haskell.

Lola specifications consist of a set of typed input and output streams that represent the input events observed by the monitor and the intermediate observations and outputs of the monitor, respectively. Specifications are defined as

equations that declaratively describe the intended values of every output stream variable in terms of the input and output streams. The set of *stream expressions* of a given type is built from constants and function symbols as constructors, and from *offset expressions* of the form s`[`k`|`d`]` where s is a stream variable, k is an integer number and d is a default value of the type of s.

For example, offset expression `balanceA[-1|0]` represents the value of stream `balanceA` in the previous step of time with `0` as the value used at the initial instant. We define a stream `balA_ok` which checks that the `balance` of account A is always above a predefined threshold of `100` tokens:

```
input Int balanceA
output Bool balA_ok = balanceA[now] > 100
```

Given values of the input streams, the formal semantics of a Lola specification is defined denotationally as the unique collection of streams of values satisfying all equations.

One of the benefits of the extensible tool HLola is its ability to define templates for stream definitions using *static parametrization*. These templates act as abstractions, hiding specific concrete values, which are instantiated in static time by the compiler. Following the previous example, we can define a more generic version of the stream `balA_ok` as follows:

```
input Int balanceA
output Bool balA_checker <Int threshold> = balanceA[now] > threshold
output Bool balA_ok = (balA_checker 100)[now]
```

However, static parametrization cannot handle parameters whose values are discovered at runtime. The values of all parameters must be determined before the monitor starts executing. Users must ensure that the resulting specification contains a finite number of streams.

3 System Architecture

Our solution is composed of a monitor generated from an HLola specification and an external component interfacing with Tezos nodes and indexers called *adapter*. When monitors start execution, we start an adapter process in charge of receiving data from the Tezos blockchain and formatting it for the monitor input. Once the monitor is online, up and running, it can send parametrized queries to the adapter to fetch subtraces of the blockchain history. The adapter can perform complex requests to the Tezos indexer, i.e. filtering and formatting the received data before redirecting the result to the monitor. Through the use of the adapter, monitors efficiently obtain newly relevant data that was previously omitted or they can process blocks that were added to the blockchain before the monitor was launched. The adapter allows monitors to be agnostic to the

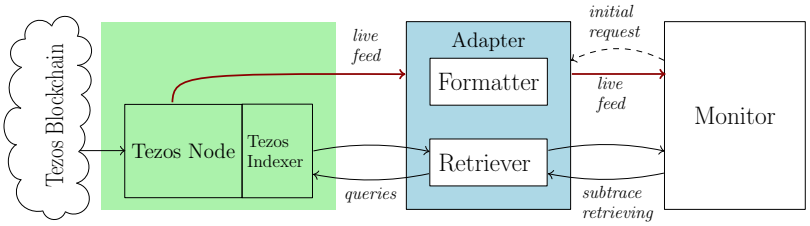

Fig. 1. Offchain monitoring system architecture

blockchain used, the indexer and the format of the data, so this architecture can be used (adapting the adapter) to other blockchains like Ethereum. Fig. 1 shows this architecture.

4 Features

In this section, we present the features of our framework and demonstrate how to apply them for effective monitoring, including to detect sandwich attacks, identify malicious and suspicious actors, and calculate attackers' profits.

4.1 Monitor Side Features

Nested monitors. Nested monitors [17] allow using SRV specifications as data functions inside other specifications. Nested specifications are created and executed dynamically. A nested monitor receives a finite subtrace of the system original trace as input, typically obtained using HLola operator s`[:`n`]`, which creates a list with the next n values of stream s.

To define a *nested* specification, we need to provide a name so we can refer to it later on and add an extra clause: `return` x `when` y where x is a stream of any type and y is a `Boolean` stream. The type of the stream x determines the type of the value returned when the specification is invoked dynamically. The Boolean stream y dictates when the nested specification finishes. The nested monitor returns the value of x at the first instant at which y becomes **true** (without needing to inspect the rest of the list), or the last value of x if y is never **true**. Nested specifications can be parametric, parameters are declared after its name. We can execute nested specifications by using the HLola function `runSpec` specifying the parameters and the input streams with which the nested monitor will be executed.

Example 1. The following specification calculates whether a transaction in the input stream `tx` is the frontrunning transaction of a sandwich attack against account `a`.

Blockchain traces may contain many trading operations involving the same pair of tokens. In a sandwich attack, the frontrunning and backrunning transactions must occur close in time to the victim transaction. For simplicity, we

consider an operation to be part of a potential sandwich attack if both the suspected frontrunning and backrunning transactions occur within 10 transactions apart from the victim transaction. We define a stream `frontruns` which first checks if tokens are traded in the transaction using function `tradeTokens`. If so, the monitor invokes the nested specification `frontrunspec` with the current transaction (`tx[now]`) as the parameter `ftx`, and the next 20 events of `tx` (`tx[:20]`) as the input stream `tx` of the nested monitor.

```
use innerspec frontrunspec
input Transaction tx
output Bool frontruns = if tradeTokens tx[now]
                        then runSpec (frontrunspec tx[now] tx[:20])
                        else False
```

We use `if · then · else ·` instead of the Boolean operator `(&&)` to stress the fact that the nested specification is only executed when tokens are traded. The nested specification `frontrunspec` defines three streams.

- Stream `counter` counts the number of transactions processed. We use it to guarantee that the victim transaction is among the first 10 transactions, and that the backrunning transaction occurs within 10 transactions of the victim transaction.
- Stream `victim` stores the position in the input trace of a potential victim transaction (a transaction where account `a` trades tokens in the same exchange as the frontrunning transaction).
- Stream `attack` indicates if an attack occurred (the client from the frontrunning transaction swapped tokens back at most 10 transactions after the victim transaction).

```
innerspec Bool frontrunspec <Transaction ftx>
input Transaction tx
const a = "tz123"
output Int counter = counter[-1|0] + 1
output Int victim =
  if counter[now] < 11 &&  tradeTokens tx[now]
     && exchange tx[now] == exchange ftx && client tx[now] == a
     && token1 tx[now] == token1 ftx && token2 tx[now] == token2 ftx
  then counter[now]
  else -1
output Bool attack =
  victim[now] != -1 && counter[now] < victim[now] + 11
  && tradeTokens tx[now] && exchange tx[now] == exchange ftx
  && client tx[now] == client ftx
  && token1 tx[now] == token2 ftx && token2 tx[now] == token1 ftx
return attack when attack
```

In a transaction where tokens are traded, the `client` of a transaction is the account that traded the tokens, the `exchange` of a transaction is the exchange where the trade happened, and `token1` and `token2` of a transaction are the traded tokens. We can further extend the above specification with a stream called `malicious` to track accounts that performed sandwich attacks against `a`.

```
output (Set Account) malicious = if frontruns[now]
  then insert (client tx[now]) malicious[-1|empty]
  else malicious[-1|empty]
```

Retroactive Nested monitors. Although nested monitors can detect sandwich attacks, this approach can be very inefficient. In a blockchain history, the number of times account `a` trades tokens (and can be victim of a sandwich attack) is significantly smaller than the total number of transactions where tokens are traded. This leads to a large number of unnecessary nested monitors being created. To address this inefficiency, we propose creating nested monitors only when account `a` trades tokens. This implies ignoring relevant transactions and later accessing them to search for potential frontrunning transactions. We achieve this *retroactive* search by implementing a function `pastRetriever` that invokes the adapter (see Section 3) to retrieve a specified number of past events.

The following specification checks whether the current transaction corresponds to address `a` trading tokens, and triggers the finer analysis of the surrounding transactions when necessary.

```
use innerspec tradersSpec
input Transaction tx
const a = "tz123"
output Bool attacked =
  if tradeTokens tx[now] && client tx[now] == a then
    let fRunners = runSpec ((tradersSpec e t1 t2) (pastRetriever 10))
        bRunners = runSpec ((tradersSpec e t2 t1) tx[:10]) in
        not (null (intersection fRunners bRunners))
  else False
  where e = exchange tx[now]
        t1 = token1 tx[now]
        t2 = token2 tx[now]
```

Within the 10 previous transactions, nested specification `tradersSpec` computes all accounts that made the same trade as `a`. On the subsequent 10 transactions, nested specification `tradersSpec` computes all accounts that did the opposite trade. Finally, the specification checks whether any account appears in both sets. We use `if · then · else False (&&)` to stress the fact that the nested specification is executed only when `a` trades tokens.

The nested specification `tradersSpec` identifies all accounts that trade two specific tokens in a given exchange, as follows:

```
innerspec (Set Account) tradersSpec <Account e> <Token t1> <Token t2>
input Transaction tx
output (Set Account) traders =
  if tradeTokens tx[now] && exchange tx[now] == e
     && token1 tx[now] == t1 && token2 tx[now] == t2
  then insert (client tx[now]) traders[-1|emptySet]
  else traders[-1|emptySet]
return traders when False
```

The above specification finds all transactions that are victims of a sandwich attack at most 10 transactions *after* it happens. Also, the nested monitors in this example are created, executed and destroyed only for every transaction where `a` exchanges tokens.

(Forward) Dynamic Parametrization. Since we have an efficient way of detecting sandwich attacks and malicious accounts, we can move on to identifying suspicious wallets, defined as those that interact with the malicious account performing the sandwich attack. The following specification computes all wallets that interact with a given account:

```
input Transaction tx
output (Set Wallet) fellows <Account a> =
  union (wallets a tx[now]) fellows[-1|empty]
```

The auxiliary function `wallets a tx` returns all wallets that sent tokens to account `a` during the execution of transaction `tx`. To identify all suspicious wallets, we need to instantiate the parametrized stream `fellows` with all malicious accounts. However, malicious accounts are only found after they perform a sandwich attack, which cannot be determined statically.

We can instantiate a parametric stream over values discovered dynamically while processing the input trace using the HLola operator `over`. The `over` operator takes two arguments: (1) a parametric stream `strm`, and (2) a stream `params` of sets of values. The resulting expression is a map where at any point in time the keys are the elements in `params[now]`, and the value associated to each key is the instantiation of `strm` over the key. For a complete description on how this operator is implemented in the tool HLola, see [25]. In our case, we can parametrize the parametric stream `fellows` over the values of stream `malicious`:

```
output (Set Wallets) suspicious = foldl union empty
  (elems (fellows `over` malicious))
```

Here, `elems m` returns the list of values in map `m`, and function `foldl f l` aggregates the elements in list `l` using `f` to combine them. To compute all suspicious

wallets, we join the suspicious wallets related to each malicious account. This specification follows each account independently.

When a new value is added to the set of parameters (the stream `malicious` in our case), we spawn a new monitor parametrized with the discovered value. The newly created nested monitor executes alongside the monitor that created it, as long as its associated parameters remain part of the set represented by stream `malicious`.

The nested monitors used for forward dynamic parametrization process the same events as the root monitor, but it is often the case that only some of the events are relevant to a specific parametrized stream. We can use subtracing to redefine stream `suspicious` as follows:

```
output (Set Wallets) suspicious = foldl union empty
  (elems (fellows `over` malicious `updating` (accounts tx[now])))
```

where `accounts` returns the set of all accounts involved in a transaction. The `updating` operator lets us specify the parameters of the monitors that have to process the current event.

In the example above, the monitor follows the dynamically parametrized stream once the parameter has been discovered (like in Lola2.0 [13] or in quantified event automata QEA [4]). However, monitoring a stream only after its parameter is discovered has its limitations, for example that the beginning prefix of the trace is ignored. In our example, this means that the monitor cannot discover wallets that interacted with a malicious account before the malicious account is identified, e.g. before a sandwich attacker reveals its identity.

We could still use *forward dynamic parametrization* to identify all wallets that ever interacted with malicious accounts, regardless of when the interaction happened. To achieve this, we need to follow all created accounts, tracked by stream `allaccounts`, and then, at every instant, keep only the wallets related to the malicious accounts discovered so far, using the stream `malicious`.

```
output (Set Account) allaccounts =
  union (createdAccounts tx[now]) allaccounts[-1|empty]
output (Set Wallet) suspicious = foldl union empty
  (elems (filterWithKey ismalicious
    (suspicious `over` allaccounts `updating` (accounts tx[now]))))
  where ismalicious k _ = member k malicious[now]
```

The function `filterWithKey p m` filters all the key-values in map `m` that satisfy the predicate `p`. Although this specification is correct, if most accounts are not malicious, this forward monitor follows many accounts unnecessarily.

However, as part of our infrastructure we have the node and indexer storing the past events of the trace, so we can combine *retroactive nested monitors* with *dynamic parametrization* when a new parameter is discovered, effectively implementing *retroactive dynamic parametrization*.

Retroactive Dynamic Parametrization. Retroactive dynamic parametrization [25] is a technique that allows monitors to revisit the past of the trace whenever a new parameter is discovered, initialize the parametric stream with the retrieved information and continue monitoring online from that point onward. This nested monitor behaves exactly as a forward parametrized monitor where an oracle had correctly guessed which parameters would be found to be useful and which parameters can be ignored. Retroactive dynamic parametrization is implemented by adding a new clause `withInit` to the `over` operator. This clause allows specifying an *initializer*, which initializes the nested monitor with events taken from the trace up to the current point. Typically, an initializer involves calling an external program (see Section 3) that interacts with an offline infrastructure to efficiently retrieve relevant past trace elements based on the discovered parameter.

We can use *retroactive* monitoring to only create the dynamic parameters when the corresponding account is malicious, and use the retroactive capability to inspect the past of the trace and see which wallets interacted with them in the past. We redefine the stream `suspicious` accordingly.

```
output (Set Wallets) suspicious = foldl union empty
 (elems (suspicious `over` malicious `updating` accs `withInit` initer))
 where accs = accounts tx[now]
```

The new `over` expression specifies an initializer `initer` (whose definition is not shown in the specification) that calls the adapter to retrieve the past of the corresponding parameter. The adapter uses the indexer to efficiently retrieve only the events in the past relevant to the current account.

4.2 Execution Simulation

To further analyze sandwich attacks, one could be interested in determining the profit obtained by the attacker. This requires reasoning about what would have happened if the invocations to the blockchain had been different. In our example, we are interested in comparing what did happen (in particular the legitimate exchange invocation) with what would have happened if the attack had not existed. For these questions, our monitoring infrastructure introduces a simple *simulation* framework.

The blockchain state is public and the code of smart contracts code is available, and evaluation frameworks are typically provided by the blockchain developers (using the exact same code that bakers execute). We use the official Tezos interpreter in our monitoring infrastructure to perform a small-step execution machine of alternative executions and observe the blockchain intermediate states. We have developed two basic building blocks:

- **Data crawler:** for a given set of operations (blocks or groups) the data crawler queries the blockchain extracting which contracts were involved in the set of transactions requested.

– **Simulation:** given a set of contracts and their state, execute a sequence of grouped transactions in order.

This allows monitors to simulate operations that happened in the blockchain and also to explore alternative histories. To simulate operations that happened, we first get the required information to execute the operations, that is, the invoking smart contract and its state, plus all other invoked smart contracts and their states. Since we know what happened because it is publicly available on the blockchain node, we can determine the smart contracts involved in a given transaction. Once we have the initial states of every contract involved, we can just execute one transaction at a time replicating the behavior executed by the blockchain.

If we diverge from transactions that happened, as it would happen if we are executing hypothetical scenarios, we may get into missing some contract states. To explore alternative histories, we first obtained the contracts that were called during a possible execution. Then, we perform a hypothetical execution, which may lead to the invocation of smart contracts whose storages were not fetched. We detect the address of the missing contract, add it to the list of required addresses for execution using the data crawler and iterate until we finish execution.

We can extend our running example about detecting sandwich attacks computing the damage suffered by the victim simulating an alternative execution in which the sandwich attack does not exist and comparing the hypothetical and the real balance of the victim.

```
input Transaction tx
input Double balanceA
const a = "tz123"
output Double stolen =
  if attacked[now]
  then (getBalance a (simulate txs')) - balanceA[now]
  else 0
  where txs' = filter (not frontruns) (pastRetriever 10)
        frontruns t = client t /= a && exchange t == exchange tx[now]
        && tokens1 t == tokens1 tx[now] && tokens2 t == tokens2 tx[now]
```

In the example above, we only expose the difference between the real and alternative balances of the victim. To precisely quantify how much the attacker stole, the monitor can also inspect manually the valuable items (as tokens) and describe the computation as an arithmetic expression. In summary, previous specifications detect and extract transactions causing sandwich attacks, filter them and observe the state of the blockchain as if these transactions had not been executed.

Number of monitored blocks	50,000
Number of transactions in monitored blocks	2,624,594
Number of transactions since the beginning of the blockchain	288,705,340
Number of calls to 3route v4	2,278
Number of attacks	39
Number of malicious accounts	3
Number of suspicious accounts	97
Number of suspicious wallets	5

Fig. 2. Summary of sandwich attack monitoring.

5 Case Studies

1. Detecting Sandwich Attacks. We used our framework to implement a retroactive dynamic monitor for the detection of sandwich attacks, identification of malicious addresses and quantification of the losses incurred.

The monitor receives a stream of transactions from the Tezos blockchain and searches for possible sandwich attacks. In this case study, we set the victim account `a` to an exchange aggregator smart contract known as *3route v4*.

We executed the monitor starting from block 5,200,000 in the Tezos main net until block 5,250,000. Retroactive parametrization allows us to start the monitor at any point in the blockchain and find fund transfers that happened before the monitor was launched. The table in Fig. 2 sums up the results obtained. Thanks to retroactive monitoring, we obtained the suspect accounts without analyzing every transaction. Instead of following 288 million transactions, the monitor only queries the adapter for the past transactions when a specific target is dynamically obtained, in the infrequent event where a suspicious account is found. Furthermore, the search for the frontrunning and backrunning transactions is only performed when the monitor detects a call to the exchange aggregator, which occurred only in 0,08% of the monitored transactions.

2. Clustering. In this case study, we consider that an address that performs front-running is a *malicious* address, and we mark the addresses that transferred cryptocurrency in the past to a malicious address as the potential source of funds is a *suspicious* address. We leave out of the search addresses that transferred funds to suspicious accounts.

When we start to follow indirectly related addresses, we find that they form *clusters* of heavily-interacting accounts with prominent addresses that act as interconnecting hubs between clusters. The flexibility of HLola allowed us to develop several implementations of clustering algorithms to discover the degree of suspiciousness of a wallet with respect to a malicious account based on how many times they interact (directly or indirectly), how many funds they exchange (directly or indirectly), and how many intermediaries are in their relation.

3. Juster. Juster is a decentralized application that allows Tezos users to bet on events that represent the changes of certain cryptocurrency prices within a given time interval. Users get a reward if their predictions are correct and lose their bet otherwise. For example, users can bet that the value of the Tezos cryptocurrency XTZ will rise by 10% or more in the following day. The Juster administrator opens events on which the users can bet and closes them after the betting interval ends, distributing the earnings accordingly.

We define an HLola monitor for the Juster platform assessing that:
(1) all closed events were previously open and no open event is reopened;
(2) there are less than 100 open events at any given time.
The monitor receives events tagged with an identifier `eventId` and with the kind of event which can be either `Open` or `Close`. We define the specification in HLola:

```
input EventId eventId
input Operation operation
define {EventId} open_events =
  if operation[now] == Open
    then insert(eventId[now], openevents[-1|{}])
  else if operation[now] == Close
    then delete(eventId[now], openevents[-1|{}])
  else openevents[-1|{}]
output Bool few_events = size(openevents[now]) < 100
output Bool right_order =
  (operation[now] == Close) == member(eventId[now], openevents[-1|{}])
```

4. BFS vs DFS in Tezos. Tezos is a self-amending blockchain that provides a mechanism to change its rules through regular protocol upgrades. Protocol Florence [24], modified the execution order of operations between smart contracts, switching from a breadth-first search (BFS) to a more conventional depth-first search (DFS) algorithm. This change in the execution order can potentially impact transactions outcomes. In this case study, we identified those transactions that could have behaved differently under the two execution orders.

A naive approach is to simulate each transaction under both execution orders and compare the results. However, this approach is very inefficient for the entire blockchain because simulating requires access to all invoked smart contracts and their states (see Section 4.2). Fortunately, most transactions are guaranteed to behave the same under BFS and DFS without simulation, because the execution order only affects if some smart contract is invoked twice and the order of the calls differs between execution orders. This is because the state of a contract only varies when the contract is invoked. The difference in the call order to a smart contract can be detected by inspecting the transaction call graph (a directed tree where nodes are labeled with smart contracts and edges represent calls).

Unfortunately, indexers do not store the transactions call graph, but only the call sequence. For each transaction, the monitor creates all possible call graphs that can generate the given call sequence when traversed with the corresponding execution order. If in one of the call graphs, the call order for a smart contract differs between BFS and DFS, the monitor marks that the transaction must be simulated.

For this case study, we considered all 34,856,986 transactions corresponding to the years 2021 and 2022. We used the adapter to retrieve from the indexer only those transactions in which some smart contract is called more than once, obtaining 1,260,145 transactions. For each transaction, then the adapter produces only its identifier, call sequence, and the execution order used when executing it. As the actual name and address of the smart contract invoked is irrelevant in this case, to save space, the formater assigns to each smart contract in a given transaction a unique small identifier. Finally, the monitor received all 1,260,145 transactions and detected that only 599,684 (out of 34,856,986) require simulation to determine behavioral differences under the other execution order.

6 Conclusions

We presented in this paper a framework for the offchain runtime verification of blockchains, and more specifically, for the Tezos Blockchain. Offchain monitoring allows us to create monitors which receive new blocks (as in online monitoring) and can perform retroactive queries to the past of the blockchain (as in offline monitoring). The retroactive feature is useful both for requesting information about the past, before the monitoring was created, and to lazily evaluate events that most of the time are irrelevant for the monitor.

We described our implementation based on stream runtime verification, and in particular on the HLola language, and several cases studies including the detection of sandwich attacks. Future work includes more advanced case studies and more quantitaitve evalutaion and comparison with other frameworks, which was beyond the scope of this work. Additionally, we plan to make the monitoring front-end available as a service, enabling its application for other blockchains.

References

1. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Proc. of ISoLA. LNCS, vol. 12478, pp. 9–24. Springer (2020)
2. Annenkov, D., Nielsen, J.B., Spitters, B.: ConCert: a smart contract certification framework in Coq. In: Proc. CPP'20. pp. 215–218. ACM (2020)
3. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: ContractLarva and open challenges beyond. In: Proc. of RV'18. LNCS, vol. 11237, pp. 113–137. Springer (2018)
4. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Proc of FM'12. LNCS, vol. 7436, pp. 68–84. Springer (2012)

5. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457. Springer (2018)
6. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a framework for certifying Tezos smart contracts. arXiv **abs/1909.08671** (2019), http://arxiv.org/abs/1909.08671
7. Bhargavan, K., Delignat-Lavaud, A., Fourneta, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Béguelin, S.Z.: Formal verification of smart contracts: Short paper. In: Proc. of PLAS@CCS'16. pp. 91–96. ACM (2016)
8. Capretto, M., Ceresa, M., Sánchez, C.: Transaction monitoring of smart contracts. In: Proc of RV'22. LNCS, vol. 13498, pp. 162–180. Springer (2022)
9. Ceresa, M., Gorostiaga, F., Sánchez, C.: Declarative stream runtime verification (hLola). In: Proc. of APLAS'20. LNCS, vol. 12470, pp. 25–43. Springer (2020)
10. Conchon, S., Korneva, A., Zaïdi, F.: Verifying smart contracts with Cubicle. In: Proc. of FMBC'19. LNCS, vol. 12232, pp. 312–324. Springer (2019)
11. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime monitoring of synchronous systems. In: Proc. of TIME'05. pp. 166–174. IEEE CS Press (2005)
12. Ellul, J., Pace, G.J.: Runtime verification of Ethereum smart contracts. In: Proc. of EDCC'18. pp. 158–163. IEEE Computer Society (2018)
13. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Hazem, T.: StreamLAB: Stream-based monitoring of cyber-physical systems. In: Proc. of CAV'19. LNCS, vol. 11561, pp. 421–431. Springer (2019)
14. Goodman, L.M.: Tezos – a self-amending crypto-ledger. https://www.tezos.com/whitepaper.pdf (2014)
15. Gorostiaga, F., Sánchez, C.: Striver: Stream runtime verification for real-time event-streams. In: Proc. of RV'18. LNCS, vol. 11237, pp. 282–298. Springer (2018)
16. Gorostiaga, F., Sánchez, C.: HLola: a very functional tool for extensible stream runtime verification. In: Proc. of TACAS'21. LNCS, vol. 12652, pp. 349–356. Springer (2021)
17. Gorostiaga, F., Sánchez, C.: Nested monitors: Monitors as expressions to build monitors. In: Proc. of RV'21. LNCS, vol. 12974, pp. 164–183. Springer (2021)
18. Gorostiaga, F., Sánchez, C.: Stream runtime verification of real-time event streams with the Striver language. STTT **23**, 157–183 (2021)
19. Gorostiaga, F., Sánchez, C.: Monitorability of expressive verdicts. In: Proc. of NFM'22. LNCS, vol. 13260, pp. 693–712. Springer (2022)
20. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebr. Progr. **78**(5), 293–303 (2009)
21. Li, A., Choi, J.A., an. Long: Securing smart contract with runtime validation. In: Proc. of ACM PLDI'20. pp. 438–453. ACM (2020)
22. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf
23. Nehaï, Z., Bobot, F.: Deductive proof of industrial smart contracts using Why3. In: Proc. of FMBC'19. LNCS, vol. 12232, pp. 299–311. Springer (2019)
24. Nomadic Labs: Protocol florence, https://tezos.gitlab.io/protocols/009_florence.html, Accessed: 2024-06-06
25. Pedregal, P., Gorostiaga, F., Sánchez, C.: A stream runtime verification tool with nested and retroactive parametrization. In: Proc. of RV'23. LNCS, vol. 14245, pp. 351–362. Springer (2023)

26. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: Safety verification of smart contracts. In: Proc. of S&P'20. pp. 1661–1677. IEEE (2020)
27. Phil, D.: Analysis of the dao exploit (2016), https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
28. Pnueli, A.: The temporal logic of programs. In: Proce. of FOCS'77. pp. 46–67. IEEE CS Press (1977)
29. Schiffl, J., Ahrendt, W., Beckert, B., Bubel, R.: Formal analysis of smart contracts: Applying the KeY system. In: Deductive Soft. Verif.: Future Perspectives - Reflections on the Occasion of 20 Years of KeY, LNCS, vol. 12345, pp. 204–218. Springer (2020)
30. Sergey, I., Kumar, A., Hobor, A.: Scilla: a Smart Contract Intermediate-Level LAnguage. CoRR **abs/1801.00687** (2018), http://arxiv.org/abs/1711.03829
31. Stephens, J., Ferles, K., Mariano, B., Lahiri, S., Dillig, I.: SmartPulse: Automated checking of temporal properties in smart contracts. In: Proc. of S&P'21. IEEE (2021)
32. Szabo, N.: Smart contracts: Building blocks for digital markets. Extropy **16** (1996)
33. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)