University of Camerino

School of Science and Technology

Master's Degree in Computer Science (LM-18)

Application of Process Mining to Blockchain-based Applications

Student
Yuri Paoloni

Supervisor
Andrea Morichetta

Co-Supervisor
Barbara Re

A.Y. 2021/2022

Abstract

With the introduction of smart contracts, blockchain has become a technology for managing trustless peer-to-peer exchanges of digital assets, paving the way for new forms of trade and business. In such a scenario, the application of process mining can help understand processes and their actual execution in ways other data analysis strategies cannot. However, applying process mining to blockchain data is challenging. The techniques created so far by researchers to address the challenge have limitations when applied to data produced by smart contracts. Some are tailored to specific use cases, such as blockchain-based Business Process Management Systems. Others focus on blockchain events, which lack a standard format and require additional processing steps. To solve this challenge, an application-agnostic extraction methodology is proposed in this thesis to collect data from every EVM-compatible smart contract and enable the application of process mining techniques. The proposed methodology aims to extract XES logs from blockchain transactions and their internal transactions. The former, unlike events, have standard parameters that ensure a common ground of operations. The latter represents the execution flow of transactions, which is the invocations of the functions of smart contracts involved in the process. The methodology includes five steps: (i) extraction of data from smart contracts, (ii) cleaning of raw data, (iii) selection of sorting criteria, (iv) construction of traces, and, finally, (v) generation of the XES log. An in-depth case study of Decentraland, a metaverse and digital assets marketplace developed on the Ethereum blockchain, has been carried out to demonstrate the validity of the proposed methodology. We were able to generate XES logs considering one or more Decentraland smart contracts combined, e.g., LANDRegistry, ESTATERegisty, and Marketplace. Unlike event data collection, extracting logs from transactions does not require prior knowledge of smart contract code. Several process mining techniques, such as process discovery and conformance checking, were applied to the generated XES logs to demonstrate the validity of the methodology.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

In recent years, blockchain has been acquiring attention as an innovative technology to execute business processes and regulate the interactions between the involved parties. The use of smart contracts makes it possible to execute complex processes (i.e., involving many business parties) in a trustless environment without the need to rely on a trusted party. Such a scenario paved the way for new forms of trade and business: exchange of digital assets, track of physical goods, complex business interactions, and more.

All these activities become appealing for process and data flow analysis through techniques like process mining. The latter have been vastly employed on traditional systems to exploit and deeply understand business processes. Examples are: producing high-level models from process logs (i.e., process discovery); comparing a process's actual execution with the intended one (i.e., conformance checking); analyzing business parties' interactions (i.e., social network analysis). In theory, it should be possible to achieve similar insights by applying process mining to business processes executed on the blockchain. However, extracting XES[9] logs from such a context is challenging. Blockchain data is not natively suitable for process mining: (i) the notion of "trace" is not present (i.e., there isn't a process identifier that groups a set of transactions together); (ii) timestamps are not accurate because transactions mined in the same block have the same value. In recent years, researchers have proposed multiple techniques to ease the extraction of logs from blockchains. Yet, each technique presents limitations that prevent the application to every smart contract. For instance, [15] is tailored to blockchain-based Business Process Management Systems. This kind of system is generated from a BPMN model. Hence, the context, the activities, and the execution flow of the process under analysis are known a priori. These assumptions do not hold for every smart contract in the blockchain. Other techniques, like [10] and [19], focus on blockchain events which are logs issued during the execution of smart contract functions. Blockchain events don't have standard parameters and are not mandatorily present in a function body. Process mining techniques work by searching for patterns and relationships among data. The lack of standard parameters makes it complicated to group blockchain events into traces using a shared parameter (i.e., case ID).

To solve this challenge, an application-agnostic extraction methodology is proposed to collect data from every EVM-compatible smart contract and enable the application of process mining techniques. The proposed methodology focuses on blockchain transactions and their internal transactions. The transactions, unlike events, have standard parameters that ensure a common ground of operations. In every transaction, there is guarantee that the *hash*, the *sender*, the *receiver*, and other parameters are present. The internal transactions represent the execution flow of transactions, which is the invocations of the functions of the smart contracts involved in the procedure.

The methodology comprises five steps:

- *Data extraction*: the smart contracts addresses and the block range are selected for the data retrieval.
- *Raw data cleaning*: hexadecimal transaction fields are converted into a readable format and unnecessary fields are removed.
- *Sorting criteria selection*: one or more of the available transaction fields are selected as sorting criteria.
- *Traces construction*: a transaction field is selected as case ID to construct traces.
- *XES log generation*: according to the selected parameters, the XES log is generated.

An in-depth case study of Decentraland, a metaverse and digital assets marketplace developed on the Ethereum blockchain, has been carried out to demonstrate the validity of the proposed methodology. Decentraland was chosen because it is an innovative and complex system that includes a wide variety of analyzable functions. Metaverses include cryptocurrencies and NFTs exchange, virtual lands, games, and more, which are usually part of different applications (e.g., NFTs marketplaces, cryptocurrency exchanges) and not combined into one. Through the methodology, it was possible to generate XES logs from different block ranges and one or more Decentraland smart contracts combined. Several process mining techniques, such as process discovery and compliance checking, were applied to the generated XES logs. Special attention was given to the logs generated from the internal transactions on which applied BPMN Miner [7], a technique able to output a BPMN model with sub-processes, boundary events, and activity markers inside.

In the following, first, relevant background information about the blockchain, Ethereum, process mining, and XES logs are introduced in Chapter 2. The proposed methodology is presented in Chapter 3, implemented in Chapter 4, and evaluated in Chapter 5. The related work is described in Chapter 6. Finally, conclusions and future challenges are listed in Chapter 7.

2. Background

This chapter aims to introduce relevant background information on the blockchain, Ethereum, process mining, and XES logs that will help the reader understand the subsequent chapters.

2.1 Blockchain

A blockchain ledger is a growing list of records, called blocks, that are securely linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. As each block contains information about the previous one, they form a chain, with each additional block reinforcing the ones before it. Therefore, blockchains are append-only and resistant to modification of their data because, once recorded, the data in any given block cannot be altered retroactively without altering all subsequent blocks. A blockchain system consists of the following components:

- A blockchain network of machines, also called nodes. There are full nodes (i.e., miners) that hold a full replica of the ledger and light nodes that send transactions and are not required to hold a full replica.
- A blockchain data structure for the ledger replicated across the blockchain network.
- A network protocol that defines rights, responsibilities, and means of communication, verification, validation, and consensus across the nodes in the network. It includes ensuring authorization and authentication of new transactions, mechanisms for appending new blocks, incentive mechanisms (if needed), and similar aspects.

In a blockchain, the parties proposing a transaction will add it to a pool (i.e., buffer) of transactions intended to be recorded on the ledger. Full nodes within the blockchain system take some of those transactions, check their integrity, and record them in new blocks on the chain. The contents of the blockchain ledger are replicated across many geographically distributed processing nodes that jointly operate the blockchain system without the central control of any single trusted third party.

There are three kinds of blockchains: public, private, and consortium. A public blockchain is a blockchain system that has the following characteristics: (i) it has an open network where nodes can join and leave as they please without requiring permission from anyone; (ii) all full nodes in the network can verify each new piece of data added to the data structure, including blocks, transactions, and effects of transactions; (iii) its network protocol includes an incentive mechanism that aims to ensure

the correct operation of the blockchain system including that valid transactions are processed and included in the ledger and that invalid transaction are rejected. It is different from a private blockchain that is usually used within a large enterprise or in a consortium of companies. In such scenarios, all blockchain nodes might be known and governed by other organizational or contractual mechanisms. These applications can be served by adopting a more relaxed trust assumption. Of course, there are no incentive mechanisms.

The most used cryptography technique in blockchains is public-key cryptography, also called asymmetric cryptography. It uses pairs of keys: public keys that may be disseminated widely (wallet address); private keys that are known only to the owner and should be kept secure for effective security (key used for sending transactions). Keys are created in pairs from a large random number using a key-generation program. In cryptocurrencies, private keys are used to manage wallets and perform operations on the chain. Lost or stolen private keys mean loss of assets.

Example of public-key cryptography: Bob encrypts the message with Alice's public key to be sure that Alice will be the only one able to decrypt it with her private key. To add also info about the sender, a signature is added. Hence, Bob signs the message with his private key so that Alice (or even anyone in the chain) can verify the signature with Bob's public key.

Users operate on blockchains using wallets which are software that allows the users to maintain the key pairs used to authenticate the transactions and verify signatures.

2.1.1 Ethereum

Ethereum[6] is a decentralized, open-source blockchain with smart contract functionality. Ethereum is public and permissionless: participants can enter and leave as they please, and the ledger data is available and visible to anyone. Ether (ETH) is the cryptocurrency generated by the Ethereum protocol as a reward to miners in a proof-of-work system for adding blocks to the blockchain. Ethereum was the first blockchain to include computational capabilities with smart contracts. Smart contracts are programs stored on a blockchain that run when predetermined conditions are met. They are typically used to automate the execution of an agreement so that all the participants can be immediately certain of the outcome without an intermediary's involvement or time loss. They can also automate a workflow by triggering actions when specific conditions are reached. Smart contracts are written with the Solidity programming language and executed on the Ethereum Virtual Machine (EVM), similar to common programming languages. Ethereum is the main blockchain for smart contract development.

2.1.2 Transactions

Transactions are cryptographically signed instructions from accounts sent on the blockchain. An account will initiate a transaction to update the state of the blockchain network. The simplest transaction is the transferring of cryptocurrencies (e.g., BTC, ETH) from one account to another. A more complex example of a transaction is the execution of a smart contract function. The transactions which involve smart contracts are executed by miners through the Ethereum Virtual Machine (EVM), which, similarly to the Java Virtual Machine (JVM), converts the transaction instructions (i.e., Solidity code) into bytecode to execute them. Once the execution is completed, the transaction is added to a block and propagated in the network. In the Ethereum

blockchain, a submitted transaction includes the following information:

- *hash*: transaction hash
- *block number*: the number of the block in which the transaction was included
- *timestamp*: time at which the transaction has been added in a block
- *to*: the receiving address (if an externally-owned account, the transaction will transfer value. If a contract account, the transaction will execute the contract code)
- *from*: the identifier of the sender. This is generated when the sender's private key signs the transaction and confirms the sender has authorized this transaction
- *value*: the amount of ETH to transfer from sender to recipient (in WEI, a denomination of ETH)
- *data*: optional field to include arbitrary data (e.g., description of the transaction, binary code to create a smart contract, function invocation)
- *gasLimit*: the maximum amount of gas units that can be consumed by the transaction. Units of gas represent computational steps
- *maxPriorityFeePerGas*: the maximum amount of gas to be included as a tip to the miner
- *maxFeePerGas*: the maximum amount of gas willing to be paid for the transaction (inclusive of baseFeePerGas and maxPriorityFeePerGas)

2.1.3 Internal transactions

An internal transaction is the consequence of smart contract logic triggered by an external transaction (i.e., regular transactions described in Section 2.1.2). A smart contract engagement can result in tens or hundreds of internal transactions. These represent value transfers occurring when a smart contract or a token transfer is executed. Internal transactions, unlike regular transactions, lack a cryptographic signature and are typically stored off-chain, meaning they are not a part of the blockchain itself. Internal transactions can be retrieved by recording all the value transfers that took place as part of external transaction execution. The following is an example of the data obtainable for internal transactions using EthTx[1] which is an advanced decoder for blockchain transactions developed by TokenFlow[2].

```
{
   ''LOAD_ID'': ''2022-05-09 14:57:50.000'',
   ''CHAIN_ID'': ''mainnet'',
   ''BLOCK'': 12010461,
   ''TIMESTAMP'': ''2021-03-10 11:14:20.000'',
   ''TX_HASH'': ''0x15ee97483fdada5e3ec49991f4de338e554c18b44a59fb3612d84faf99b52dcd
      '',
   ''CALL_ID'': ''\\N'',
   ''CALL_TYPE'': ''call'',
   ''FROM_ADDRESS'': ''0x28cd504f564c7288413148a78788e29e7fffebf0'',
   ''FROM_NAME'': ''0x28cd504f564c7288413148a78788e29e7fffebf0'',
   ''TO_ADDRESS'': ''0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d'',
```

[1] https://ethtx.info/
[2] https://tokenflow.live/

```
12     ``TO_NAME'': ``LAND'',
13     ``FUNCTION_SIGNATURE'': 22465,
14     ``FUNCTION_NAME'': ``setApprovalForAll'',
15     ``VALUE'': 0,
16     ``ARGUMENTS'': ``{\``address\'':\``0x8e5660b4ab70168b5a6feea0e0315cb49c8cd539
           \'',\``approved\'':\``True\``}'',
17     ``RAW_ARGUMENTS'': ``[{\``name\'':\``address\'',\``raw\'':\``0
           x0000000000000000000000008e5660b4ab70168b5a6feea0e0315cb49c8cd539\'',\``type
           \'':\``address\``},{\``name\'':\``approved\'',\``raw\'':\``0
           x0000000000000000000000000000000000000000000000000000000000000001\'',\``type
           \'':\``bool\``}]'',
18     ``OUTPUTS'': ``{}'',
19     ``RAW_OUTPUTS'': ``[]'',
20     ``GAS_USED'': 29105,
21     ``ERROR'': null,
22     ``STATUS'': true,
23     ``ORDER_INDEX'': 1315,
24     ``DECODING_STATUS'': true,
25     ``STORAGE_ADDRESS'': ``0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d``
26 }
```

The *CALL_ID* field permits distinguishing a regular transaction from an internal transaction. When it is *"\\N"* it indicates that the record is a transaction. Instead, for sub-calls, it can assume different values indicating deeper levels in the trace. Example are: *"0"*, *"0_0"* (sub-call of 0), *"0_1"* (on the same level of *"0_0"*), *"0_0_0"*, *"0_2"*, *"0_2_0"*, etc...

Putting together the internal transactions, it is possible to obtain the full execution trace of a regular transaction. Such data can be processed using process mining to get details about process variants. For instance, an infrequent variant could indicate a bug in the immutable code.

2.1.4 Events

Events are logs issued during the execution of smart contract functions. An event hasn't standard parameters and is not mandatorily present in a function body. The smart contracts developers add the events where they need to log something out. In general, events log the outcome of an operation (e.g., transfer of a token, deposit). Logged data is used by external services, like frontends, to update their internal states accordingly. The following is a simple Solidity code example that shows the use of events to log that the value has changed.

```
1  pragma solidity 0.5.17;
2
3  contract Counter {
4
5     event ValueChanged(uint oldValue, uint256 newValue);
6
7     // Private variable of type unsigned int to keep the number of counts
8     uint256 private count = 0;
9
10    // Function that increments our counter
11    function increment() public {
12       count += 1;
13       emit ValueChanged(count - 1, count);
14    }
15
16    // Getter to get the count value
17    function getCount() public view returns (uint256) {
18       return count;
19    }
20
21 }
```

2.2 Process mining

Process mining is a family of techniques relating to the fields of data science and process management to support the analysis of operational processes based on event logs. The goal of process mining is to turn event data into insights and actions. Process mining is an integral part of data science, fueled by the availability of event data and the desire to improve processes. Process mining techniques use event data to show what people, machines, and organizations are doing. Process mining provides novel insights that can be used to identify the execution path taken by operational processes and address their performance and compliance problems.

Process mining starts from event data. Input for process mining is an event log. An event log views a process from a particular angle. Each event in the log should contain (i) a unique identifier for a particular process instance (called case id), (ii) an activity (description of the event that is occurring), and (iii) a timestamp. There may be additional event attributes referring to resources, costs, etc., but these are optional. With some effort, such data can be extracted from any information system supporting operational processes. Process mining uses these event data to answer a variety of process-related questions. There are three main classes of process mining techniques: process discovery, conformance checking, and process enhancement.

2.2.1 Process discovery

A process discovery algorithm is a function that maps an event log onto a process model such that the model is representative of the behavior seen in the event log. In recent years, researchers have proposed multiple discovery algorithms. Each algorithm differs in the internal process, the properties, and the utilized graph (e.g., Petri nets, directly-follows graphs). The results of process discovery algorithms can be evaluated according to four quality criteria:

- Fitness: the discovered model should be representative of the behavior seen in the log. The higher the fitness, the higher the number of traces present in the log represented in the discovered model.

- Precision: the discovered model should not represent completely unrelated behaviors to what was seen in the event log. It is related to the notion of underfitting, for which a model having poor precision is underfitting, i.e., it represents behavior that is very different from what was seen in the event log.

- Generalization: the discovered model should generalize the example behavior seen in the event log. It refers to how the log and the process model match. It is related to the notion of overfitting, for which an overfitting model does not generalize enough, i.e., it is too specific and too driven by the event log.

- Simplicity: the discovered model should be as simple as possible. It takes into account the number of elements in the discovered model.

The most common algorithms are the Alpha algorithm[1], the Heuristic Miner [21], and the Inductive Miner[11].

2.2.2 Conformance checking

Conformance checking relates the events in the event log to the activities in the process model and compares both. The goal is to find commonalities and discrepancies between the modeled behavior and the observed one. Conformance checking is relevant for business alignment and auditing to find undesirable deviations suggesting fraud or inefficiencies, measuring the performance of process discovery algorithms, and repair models that are not aligned well with reality. Two main approaches for conformance checking:

- Token-based replay matches a trace and a Petri net model, starting from the initial place, in order to discover which transitions are executed and in which places we have remaining or missing tokens for the given process instance.
- Alignment-based replay aims to find one of the best alignments between the trace and the model. For each trace, the output of an alignment is a list of couples where the first element is an event (of the trace) or and the second element is a transition (of the model) or . Three kinds of couples: sync move (i.e., both trace and model advance), move on log (i.e., move in the trace that is not mimicked in the model), and move on the model (i.e., move in the model that is not mimicked in the trace).

2.3 XES logs

The XES [9] standard is officially published by IEEE. The XES standard defines a grammar for a tag-based language to provide a unified and extensible methodology for capturing event logs and event streams. An XML Schema for event logs and extensions is part of the standard. An XES instance corresponds to a file-based event log that can be used to transfer event-driven data from a site generating this event-driven data to a site where this data will be analyzed. The primary purpose of XES is process mining.

An XES file is structured as follows. On the top level, there is one **log** object (line 1) which contains all event information related to one specific process. A log contains an arbitrary number of **trace** objects (line 4-21). Each trace describes the execution of one specific instance, or case, of the logged process. Every trace contains an arbitrary number of event **objects** (line 6-12, 13-20). Events represent atomic granules of activity that have been observed during the execution of a process. The log, trace, and event objects contain no information themselves. They only define the structure of the document. All information in an event log is stored in attributes. All attributes have a string-based key (lines 7-11, 14-19).

```
1  <log xes.version=``1.0'' xes.features=``nested-attributes'' openxes.version=``1.0RC7
       '' xmlns=``http://www.xes-standard.org/''>
2        <string key=``concept:name'' value=``TEST''/>
3        <string key=``lifecycle:model'' value=``standard''/>
4        <trace>
5              <string key=``concept:name'' value=``1411''/>
6              <event>
7                    <string key=``id'' value=``1411''/>
8                    <string key=``org:resource'' value=``user''/>
9                    <date key=``time:timestamp'' value=``1970-03-27T12:02:00+10:00''/>
10                   <string key=``lifecycle:transition'' value=``complete''/>
11                   <string key=``concept:name'' value=``A''/>
12             </event>
13             <event>
14                   <string key=``id'' value=``1411''/>
```

```
15                  <string key=``org:resource'' value=``user''/>
16                  <date key=``time:timestamp'' value=``1970-03-27T12:32:00+10:00''/>
17                  <string key=``lifecycle:transition'' value=``complete''/>
18                  <string key=``concept:name'' value=``S1A''/>
19                  <string key=``ids1'' value=``2S2505''/>
20              </event>
21      </trace>
22  </log>
```

3. Methodology

The analysis of the existing research highlighted the lack of an application-agnostic methodology to extract process logs from blockchains and enable the application of process analysis techniques on blockchains easily. An extraction methodology capable of collecting data from every smart contract and generating XES logs ready for techniques like process mining without requiring complex configurations. To work with every smart contract, the methodology needs data with standard parameters, which, unlike events, gives a common ground for operations. Blockchain transactions, as described in Chapter 2, have standard parameters and, for this reason, have been selected as the input data of the methodology. Thanks to transactions, the methodology can rely on the *hash*, *from*, *to*, and other transaction fields being always present in the data under the collection. Blockchain transactions with smart contract addresses as receivers are an invocation of smart contract functions. This means that the extraction process leads to a process log with events representing smart contract functions. In addition to transactions, the methodology will focus on internal transactions to exploit the execution trace of smart contract functions through process mining. The smart contract code is immutable and cannot be changed as quickly as in traditional systems, so it is critical to be able to understand how smart contracts behave and intervene early to prevent problems. The analysis of internal transactions returns information on the smart contract functions internal execution, including parameters, return values, and more that can help understand how smart contracts are actually performing. Internal transactions have been as well described in Chapter 2.

The methodology has been implemented to deal with three different data inputs:

- **transactions**: collect transactions from one or more smart contracts and generate a XES log according to the selected parameter: *sort by*, *case ID*, and *concept name*.
- **internal transactions**: collect internal transactions from one or more smart contracts and generate an XES log for each different top-level function name. Each process log refers to a specific function, and its traces, which contain internal executions, are grouped by transaction hash.
- **combination of the transactions and internal transactions**: combine transactions and the respective internal transactions to obtain logs that contain sub-processes. To exploit these logs, a dedicated technique is needed. This technique is BPMN Miner[7], a discovery algorithm that can produce a BPMN model with sub-processes, boundary events, and activity markers inside.

The proposed methodology, as depicted in the figure 3.1, consists of five steps described in the following sections: (i) *data extraction*, (ii) *raw data cleaning*, (iii) *sorting criteria selection*, (iv) *trace construction*, and (v) *XES generation*.

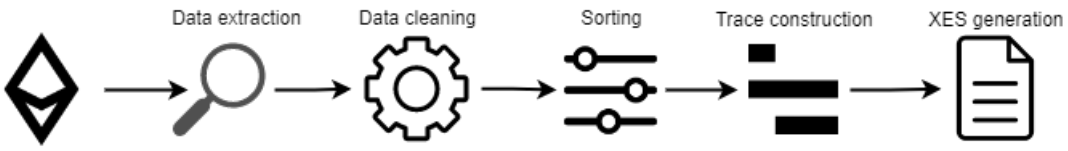

Figure 3.1: Overview of the methodology

3.1 Data extraction

The *data extraction* step, shown in figure 3.2, is the first step of the methodology. As the name implies, blockchain data is extracted according to two parameters: smart contract addresses and block range. The former indicates the addresses from which to collect transaction data. The latter is optional and is used to scope the research in a precise block number interval. More specifically, the methodology requires two addresses for each smart contract: a transaction address from which it will collect transactions, and an ABI address which will be used to get the Application Binary Interface (ABI)[1] of the contract. The ABI is the interface of the smart contract and is the de facto method for encoding Solidity smart contract calls for the Ethereum Virtual Machine (EVM) and backward. The ABI is required to decode hexadecimal fields in the *raw data cleaning* step. In most cases, the transaction address and the ABI address coincide, except when a smart contract is developed with the proxy pattern[2] to allow the upgradability of the smart contract. A proxy architecture is such that all message calls go through a Proxy contract that will redirect them to the latest deployed contract logic (i.e., ProxyContract→Contract). This means that the methodology should collect transactions from the ProxyContract (e.g., LANDProxy[3]) and use the ABI of the main contract (e.g., LANDRegistry[4]) to decode the hexadecimal fields.

The blockchain data that the methodology is going to collect are transactions and the respective internal transactions. The reasons behind the usage of transactions as input data have been explained in the previous sections.

3.1.1 Transactions

The collected transactions are in the format described in Section 2.1.2. Transactions can be collected using blockchain explorers like Etherscan [5]. The Algorithm 1 shows the code for transactions collection.

Algorithm 1 Transactions retrieval

Require: $contracts\_addresses, start\_block, end\_block$

1: **for all** *address* in $contracts\_addresses$ **do**
2: $\quad transactions = etherscan.get\_txs\_by\_address(address, start\_block, end\_block)$
3: **end for**

[1] https://docs.soliditylang.org/en/v0.8.13/abi-spec.html
[2] https://blog.openzeppelin.com/proxy-patterns/
[3] https://etherscan.io/address/0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d
[4] https://etherscan.io/address/0xa57e126b341b18c262ad25b86bb4f65b5e2ade45
[5] https://etherscan.io/

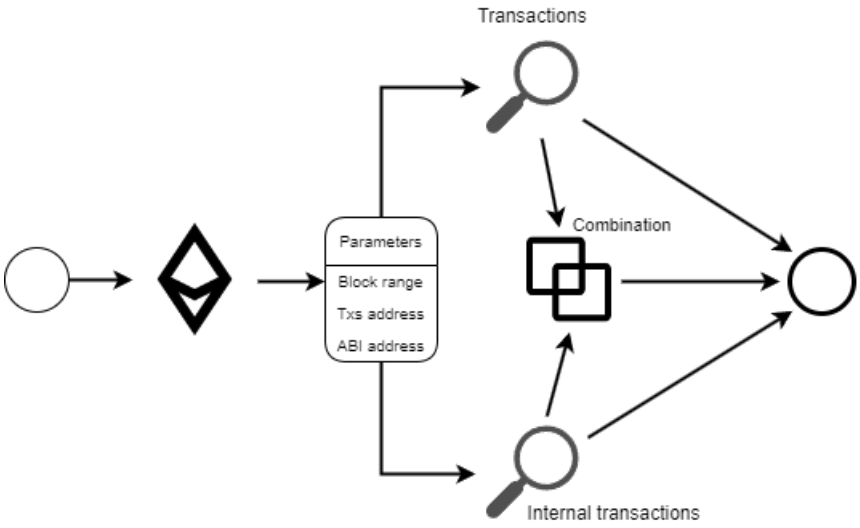

Figure 3.2: Data extraction step

3.1.2 Internal transactions

Internal transactions are not a part of the blockchain itself. They are typically stored off-chain by recording the value transfers that took place during a transaction. To collect such data, specific tools are needed. An example is EthTx[6], an advanced decoder for blockchain transactions that returns a lot of information about transactions including emitted events, tokens transfers, and the trace of internal transactions. The algorithm to collect internal transactions is similar to Algorithm 1 with the only difference of EthTx instead of Etherscan for data collection. The data obtained is in the format shown in Section 2.1.3.

3.2 Raw data cleaning

The raw data cleaning step is illustrated in figure 3.3. Some fields of the collected transactions are encoded in hexadecimal according to the smart contract Application Binary Interface (ABI). One of the encoded transaction fields is *input* which contains the executed function name and the respective parameters.

The Algorithm 2 shows the code for the input field decoding. First, the smart contract ABI is obtained (line 1) using the ABI address provided in the previous step. Then, the algorithm iterates through the transactions to decode the *input* field and store the function signature in a new field *function* and function parameters in another field *function_params* (lines 2-6).

[6] `https://ethtx.info/`

Algorithm 2 Data decoding

1: $contractABI = get\_contract\_ABI(contract)$
2: **for all** $transaction$ in $transactions$ **do**
3: $\quad function, function\_params = decode\_input(contract\_ABI, transaction.input)$
4: $\quad transaction.function = function$
5: $\quad transaction.function\_params = function\_params$
6: **end for**

In addition to the operation described above, the fields not needed in the next steps are removed: *nonce*, *value*, *isError*, *txreceipt_status*, *input*, *contractAddress*, *confirmations*, *methodId*.

The decoding step is not performed on the internal transaction since the data obtained with EthTx contains both the raw and the decoded data. Instead, unnecessary fields are removed also there: *LOAD_ID*, *CHAIN_ID*, *FUNCTION_SIGNATURE*, *VALUE*, *RAW_ARGUMENTS*, *RAW_OUTPUTS*, *ERROR*, *STATUS*, *DECODING_STATUS*, *STORAGE_ADDRESS*.

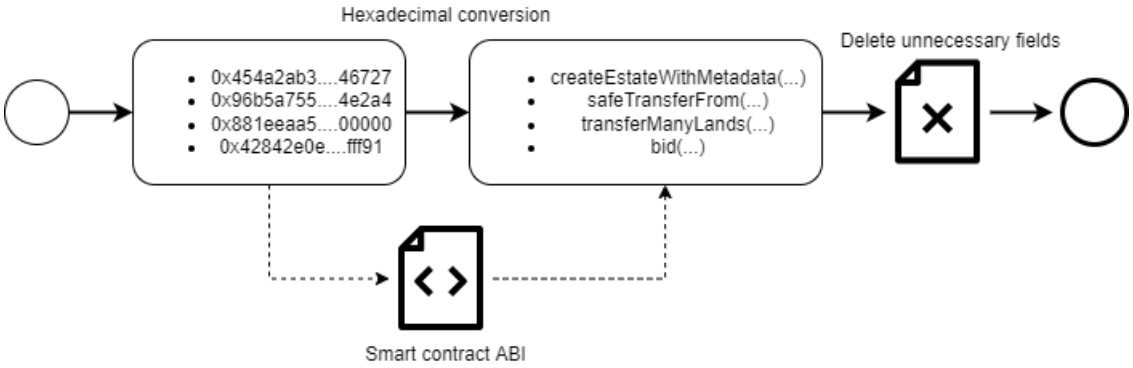

Figure 3.3: Raw data cleaning step

3.3 Sorting criteria selection

The sorting step is shared for transactions and internal transactions and is represented in figure 3.4. In this step, one or more fields from available transactions fields (i.e., non-removed standard fields plus the function parameters extracted in the *raw data cleaning* step) are selected to sort the records.

As noted in Chapter 1, timestamps are not accurate because transactions mined in the same block have the same timestamp value. In blockchains, transactions are executed when they are validated and put inside a block, not when the accounts send them. Fortunately, each transaction has the *transactionIndex* field that indicates the ordering of transactions inside a block. In general, this reflects the execution order of transactions. For this reason, the methodology, by default, sorts transactions by *timestamp* and *transactionIndex*. However, it also supports sorting by one or more selected transaction fields.

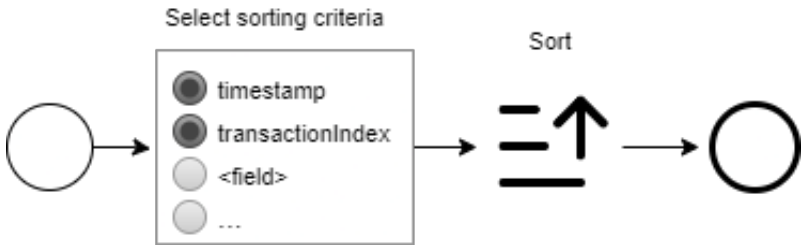

Figure 3.4: Sorting criteria selection step

3.4 Trace construction

The *trace construction* step, shown in figure 3.5, is a crucial step for the output of process analysis techniques executed on the generated XES log. The selection of the case ID strongly influences the outcome of the techniques and allows to highlight different aspects.

The trace construction algorithm differs for the three inputs, i.e., transactions, internal transactions, and a combination of the two.

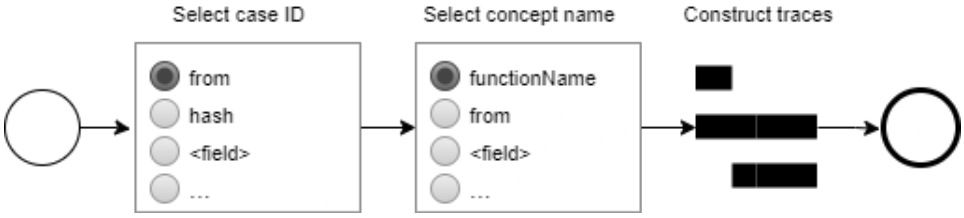

Figure 3.5: Trace construction step

3.4.1 Transactions

For the transaction trace construction phase only the case ID needs to be selected. This field will affect the result of the process mining techniques that will be applied to the generated log. For example, if the field *from* (i.e., the sender of the transaction) is selected as case ID, all transactions sent by an address (i.e., account) will be included in a trace. Performing process discovery on such a log will discover a model representing the lifecycle of transactions sent by that address to a specific contract.

Algorithm 3 Transactions trace construction

1: $transactions[\text{“}org:resource\text{''}] = transactions[\text{“}from\text{''}]$
2: $transactions[\text{“}case:concept:name\text{''}] = transactions[\text{“}from\text{''}]$
3: $transactions[\text{“}time:timestamp\text{''}] = transactions[\text{“}timeStamp\text{''}]$
4: $transactions[\text{“}concept:name\text{''}] = transactions[\text{“}inputFunctionName\text{''}]$

As shown in Algorithm 3, there isn't a complex manipulation of the records to generate traces. The case ID and other XES-standard fields are selected from the existing transaction fields (i.e., non-removed standard fields plus the function parameters extracted in the *raw data cleaning* step):

- **case:concept:name**. It groups events in traces. *case:concept:name* is the notion for case ID in XES logs.

- **concept:name**. It gives the name to XES events. For instance, in models generated with process discovery, the *concept:name* is the name of the activities.

- **time:timestamp**. It is used in XES logs and process mining techniques as the standard timestamp field.

- **org:resource**. It indicates the *resource* participating in a specific event.

The field *from* is select as *org:resource* (line 1) since it contains the address of the account sending the transaction. The field *from* is also selected as *case:concept:name* (i.e., case ID) to generate traces containing all the transactions sent by an account (line 2). The field *timeStamp* is selected as *time:timestamp* (line 3). The field *inputFunctionName* is selected as *concept:name* (line 4) to name the XES event with the smart contract function executed in the transaction.

3.4.2 Internal transactions

For internal transactions, the situation is more complex. A dedicated algorithm, shown in Algorithm 4, is needed.

As mentioned at the beginning of the chapter, the objective is to create a different log for each function name present in the input data. A function name is the name of the smart contract function performed during the execution of a transaction. The data obtained with EthTx contains both parent transactions (i.e., the actual transaction with the hash) and their respective internal transactions. Each record has a *TX_HASH* that indicates the parent transaction to which the internal transaction belongs. To create separate logs the internal transactions should be grouped into traces using the parent transaction hash. In this way, every trace will represent an execution trace of a transaction. To create different subsets for each function name the *CALL_ID* field can be used. Records with *CALL_ID* == "\\N" are the parent transactions and the records with *CALL_ID* != "\\N" are internal transactions. The methodology iterate through the records (lines 2-15) and store the sender address in correspondence of *CALL_ID* == "\\N" (line 4). In the same iteration, it stores on the internal transaction the sender address in *ORIGIN_ADDRESS* since in internal transactions the *FROM_ADDRESS* fields refer to the previous smart contract in the execution trace (line 8). It also prefix the function name of internal transactions with the name of the smart contract executing it, e.g., {*contract_name*}_{*function_name*} (line 6). This is done to identify the smart contract that performs a function, as it may happen that function names are similar among smart contracts (e.g., the function *transfer* is implemented in multiple smart contracts). The last step is the splitting of records into different sets where each contains the internal transactions for a specific function name (lines 10-15). This is done by creating a new column where the *CALL_ID* is equal to "\\N" (line 10), then grouping by *TX_HASH* and mapping the *TX_HASH* to the first value of the newly created column (i.e. *NEW_HASH _GROUP*) (line 11) as there can only be one *CALL_ID* == "\\N" per *TX_HASH*. Finally, each group is stored in a different data structure (lines 13-15).

Algorithm 4 Internal transactions trace construction

```
1: user_address = ""
2: for all (transaction, index) in transactions do
3:    if transaction["CALL_ID"] = "\\N" then
4:       user_address = transaction["FROM_ADDRESS"]
5:    else
6:       transactions[index, "FUNCTION_NAME"] =
             "{transaction["FROM_NAME"]}_{transaction["FUNCTION_NAME"]}"
7:    end if
8:    transactions[index, "ORIGIN_ADDRESS"] = user_address
9: end for
10: transactions["NEW_HASH_GROUP"] = (if transactions["CALL_ID"] =
        "\\N") then * transactions["FUNCTION_NAME"]
11: transactions["GROUP"] = transactions["TX_HASH"].map(
        transactions.groupby("TX_HASH")["NEW_HASH_GROUP"].first())
12: function_names = []
13: for all group in transactions.groupby("GROUP") do
14:    function_names.push(group)
15: end for
```

3.4.3 Combination of transactions and internal transactions

The procedure for the combination of transactions and internal transactions is shown in Algorithm 5.

As said at the start of the chapter, the end goal is to execute the BPMN Miner algorithm. In order to make the dataset conform to that algorithm, some changes to the records are required. The conditions behind BPMN Miner are similar to the notions of the key and foreign key in relational databases. It seeks to split the log into sub-logs based on process instance identifiers (i.e., keys) and references from sub-process to parent process instances (i.e., foreign keys). For an in-depth explanation of BPMN Miner refer to [7].

The input is the transactions internal execution dataset which contains both parent-transactions (i.e., CALL_ID == "\\N") and internal transactions (i.e., CALL_ID != "\\N"). The strategy is to create on the record of internal transactions a new field with the key equal to the name of the function of a parent transaction and the value equal to the hash of the parent transaction (line 21) (e.g., *"function_name": "0x....123"*). In this way, a relationship is added between internal transactions that allow BPMN Miner to group them. Moreover, the *concept name* of the records with *CALL_ID == "\\N"* is prefixed with the smart contract name (e.g., *LAND.transferLand*) (line 17), and the concept name of records with *CALL_ID != "\\N"* is prefixed with {*function_name*}_{*from*} (e.g., *approve_LAND.approve*) (line 20) to highlight the sub-process name (i.e., *function_name)* and the smart contract that is calling it (i.e., *from*).

The *id* column is added to the transaction record and to the related sub-calls to group them in traces (line 32). This column contains the address of the user invoking the "top-level" transaction. In the end, there are two groupings in the data: (i) the *id* field used to group records in traces, and the new field (i.e. the assignment done in line 21) to identify sub-processes. Moreover, since BPMN Miner allows the selection of the keys for which sub-processes are wanted, the *function_name* column on events

belonging to execution traces with length one (i.e., without internal transactions) is prefixed with FK to indicate that they should not be selected (lines 7-13 and 24-30). These modifications have been done using as reference the artificial log contained in the BPMN Miner source code[7] and also mentioned in [7].

Algorithm 5 Trace construction for BPMN Miner

```
1: user_address = ""
2: function_name = ""
3: calls_length = ""
4: columns = {}
5: for all (transaction, index) in transactions do
6:   if transaction["CALL_ID"] = "\\N" then
7:     if calls_length = 1 then
8:       if columns.get(function_name)! = function_name then
9:         columns.update({function_name : "FK_{function_name}"}
10:      end if
11:    else
12:      columns.update({function_name : function_name}
13:    end if
14:    calls_length = 0
15:    user_address = transaction["FROM_ADDRESS"]
16:    function_name = transaction["FUNCTION_NAME"]
17:    transactions[index, "FUNCTION_NAME"] = "{function_name}_
           {transaction["FROM_NAME"]}.{transaction["FUNCTION_NAME"]}"
18:  else
19:    calls_length+ = 1
20:    transactions[index, "FUNCTION_NAME"] = "{function_name}_
           {transaction["FROM_NAME"]}.{transaction["FUNCTION_NAME"]}"

21:    transactions[index, function_name] = transaction["TX_HASH"]
22:  end if
23:  if index = transactions.length − 1 then
24:    if calls_length = 1 then
25:      if columns.get(function_name)! = function_name then
26:        columns.update({function_name : "FK_{function_name}"}
27:      end if
28:    else
29:      columns.update({function_name : function_name}
30:    end if
31:  end if
32:  transactions[index, "id"] = user_address
33: end for
```

3.5 XES log generation

After the trace construction step the methodology is finally ready to generate the XES log according to parameters selected in the previous steps.

[7] https://svn.win.tue.nl/repos/prom/Packages/BPMNMiner/Trunk/log/

4. Tool implementation

This chapter describes the libraries used to implement the methodology described in 3 as a framework and client-server application. It also illustrates a demo of the application's usage.

4.1 Libraries

The methodology described in Chapter 3 has been implemented with Python and integrated into a client-server application for end users' usage. The application allows the collection of transactions and the generation of XES logs from one or more smart contracts and a block range. It is deployed at `https://xes-ethereum-extractor.herokuapp.com/`. The framework and the client-server application source code are publicly available in the project's GitHub repository[1]. The repository contains also the Python scripts, Jupyter notebooks[2], datasets, and models used during the implementation.

The figure 4.1 illustrates the components of the client-server application. In the client, there is Next.js that displays the data and makes requests to the server. The requests on the server are handled by Flask, which will then route them to the proper service, e.g., Pandas to process data, Etherscan to fetch transactions, and PM4Py to generate the XES logs. The following sections describe the libraries employed in the methodology implementation.

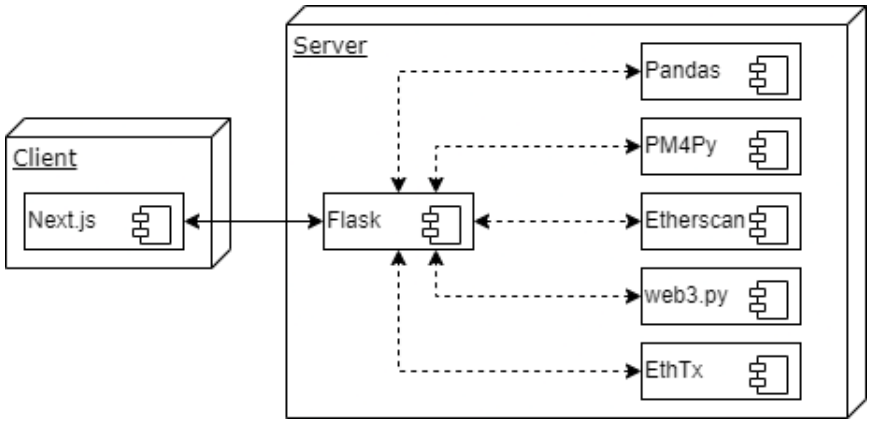

Figure 4.1: Client-server application

[1] `https://github.com/yuripaoloni/xes-ethereum-extractor`
[2] `https://jupyter.org/`

Etherscan

Etherscan[3] is a block explorer for the Ethereum network. It allows you to search through transactions, blocks, wallet addresses, smart contracts, and other on-chain data. Etherscan provides APIs and libraries[4] for developers that, in our case, were used to collect the transactions data from the Ethereum blockchain about the specified smart contract addresses and within the selected block range. Etherscan was chosen since it is the leading block explorer on the Ethereum blockchain and provides maintained Python libraries. It was used to fetch the smart contracts ABI and for the collection of transaction data, not for internal transactions.

Ethtx

EthTx Transaction Decoder[5] is an advanced decoder for blockchain transactions that translates complex transaction data into an organized and readable format that encompasses all triggered events, function calls, token movements, and more. Among the few existing tools for collecting internal transactions, EthTx was chosen because it is the most reliable, supported, maintained, and provides a Python library.

Web3.py

Web3.py[6] is the main Python library for interacting with the Ethereum blockchain. It provides several APIs to interact with smart contracts, transactions, ABIs, and more. In general, it is used in DApps to instantiate smart contracts, send transactions, and wait for a response. In our case, Web3.py has been employed in the data collection step to decode the hexadecimal fields through the smart contracts ABI.

Infura

Infura[7] provides the tools and infrastructure that allow developers to easily take their blockchain application from testing to scaled deployment with simple, reliable access to Ethereum and IPFS. Blockchain applications need connections to peer-to-peer networks that can require long initialization times. It can take hours or days to sync a node with the Ethereum blockchain and can use more bandwidth and storage than you had planned. Moreover, it can get expensive to store the full Ethereum blockchain. Etherscan and Web3.py require an Ethereum node to access the Ethereum blockchain and read the desired data. Infura was used to get a free Ethereum node avoiding the described criticalities of autonomously setting up a node.

Pandas

Pandas[8] is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers complex operations for manipulating dataframes (i.e., Pandas native data structure), such as data alignment, handling of

[3] https://etherscan.io/
[4] https://docs.etherscan.io/misc-tools-and-utilities/using-this-docs
[5] https://tokenflow.live/products/ethtx
[6] https://web3py.readthedocs.io/en/stable/
[7] https://infura.io/
[8] https://pandas.pydata.org/

missing data, reshaping and pivoting datasets, subsetting, merging, and more. Pandas were utilized in almost every step of the methodology because the data was input as a Pandas dataframe. More specifically, it was used to sort the dataset, remove or change fields, and manipulate the dataset in the trace construction step.

PM4Py

PM4Py[9] is a Python library that supports (state-of-the-art) process mining algorithms. It is completely open-source and intended to be used in both academia and industry projects The main features are:

- Handling event data: importing and exporting event logs stored in various formats;
- Filtering event data: specific methods to filter an event log based on a timeframe, case performance, start/end activities, variants, and attribute values;
- Process discovery: provides alpha miner, inductive miner, heuristic miner, directly-follows graphs, and others...
- Petri net management;
- Conformance checking: provides token-based replay and alignments
- Statistics: it is possible to calculate different statistics on top of classic event logs and data frames
- Log-Model evaluation: it is possible to compare the behavior contained in the log and the behavior contained in the model, to see if and how they match;
- Simulation: it offers different simulation algorithms, that starting from a model, can produce an output that follows the model and the different rules that have been provided by the user:

The library has been employed to generate XES logs and apply the above-listed process mining techniques to validate the methodology.

ProM

ProM[10] is an Open Source framework for process mining algorithms. ProM is an extensible framework that supports a wide variety of process mining techniques in the form of plugins. One of these plugins is BPMN Miner[7]. ProM has been utilized to test the XES log generated with the combination of transactions and their internal transactions. The results are described in Chapter 5.

Flask

Flask[11] is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction

[9] `https://pm4py.fit.fraunhofer.de/`
[10] `https://www.promtools.org/doku.php`
[11] `https://flask.palletsprojects.com/en/2.2.x/`

layer, form validation, or any other components where pre-existing third-party libraries provide common functions. Exactly for its simplicity and lack of extensive configuration, Flask was the perfect choice as the framework's server. It was used to define APIs to fetch transactions, generate the XES log and download it

Next.js

Next.js[12] is a React framework that enables several extra features, including server-side rendering and generating static websites. React[13] is a JavaScript library that is traditionally used to build web applications rendered in the client's browser with JavaScript. Developers recognize several problems with this strategy, however, such as not catering to users who do not have access to JavaScript or have disabled it, potential security issues, significantly extended page loading times, and harm to the site's overall search engine optimization. Frameworks such as Next.js sidestep these problems by allowing some or all of the website to be rendered on the server side before being sent to the client. Next.js was a perfect match since the frontend pages aren't changing much and could be rendered at build time. This way the website is fast and optimized for SEO.

Heroku

Heroku[14] is a cloud platform as a service (PaaS) supporting several programming languages. Applications that are run on Heroku typically have a unique domain used to route HTTP requests to the correct application container. Heroku is famous for its simplicity and speed of application deployment. You simply need to add a Git remote URL to the application repository to deploy. Heroku was chosen to deploy the application since it integrates well with Flask. The Flask server and Next.js client were deployed on two different application containers that communicate via HTTP REST calls.

4.2 How to use

The client-server application is deployed at `https://xes-ethereum-extractor.herokuapp.com/`. It contains three pages: the landing page, the tool page, and the about page. The landing page is shown in figure 4.2 and the about page is shown in figure 4.3. The latter contains the abstract of this elaborate to provide context and a "How to Use" section with details on how to fill out forms, generate data, and more.

[12] `https://nextjs.org/`
[13] `https://reactjs.org/`
[14] `https://www.heroku.com/`

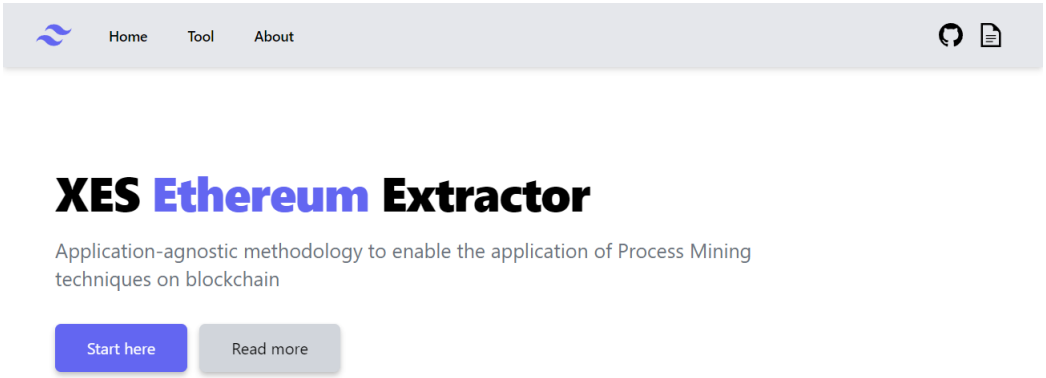

Figure 4.2: Landing page

Figure 4.3: About page

The tool page includes two steps: the transaction collection and the XES log generation. The first step is shown in figure 4.4. In this step, the user specifies the smart contracts to which collect transactions and the block range that by default is *0 - 99999999*. Regarding the smart contracts, the following information should be provided:

- *Name*: the name of the smart contract. It is not mandatory to use the actual name of the smart contract (e.g., LANDRegistry). The purpose of the name is to help identify the smart contract to which the collected transactions and generated logs belong.

- *Transaction address*: the address at which transactions will be collected.
- *ABI address*: the address of the smart contract which contains the logic to execute transactions and, therefore, also the ABI. In general, it coincides with the *transaction address* but, in some cases (e.g., Proxy pattern), it does not. An example is the LANDRegistry contract which is contacted through a proxy contract (i.e., LANDProxy).

Additional smart contracts to analyze can be added through the *"+"* button on the right of the form. Clicking on the *"SEARCH"* button will fire the transactions collection. As illustrated in figure 4.4, a preview of the results is displayed for quick reference. The full dataset can be downloaded via the *'Download full dataset"* button. The XES log generation step can be accessed via the *'Next step"* button.

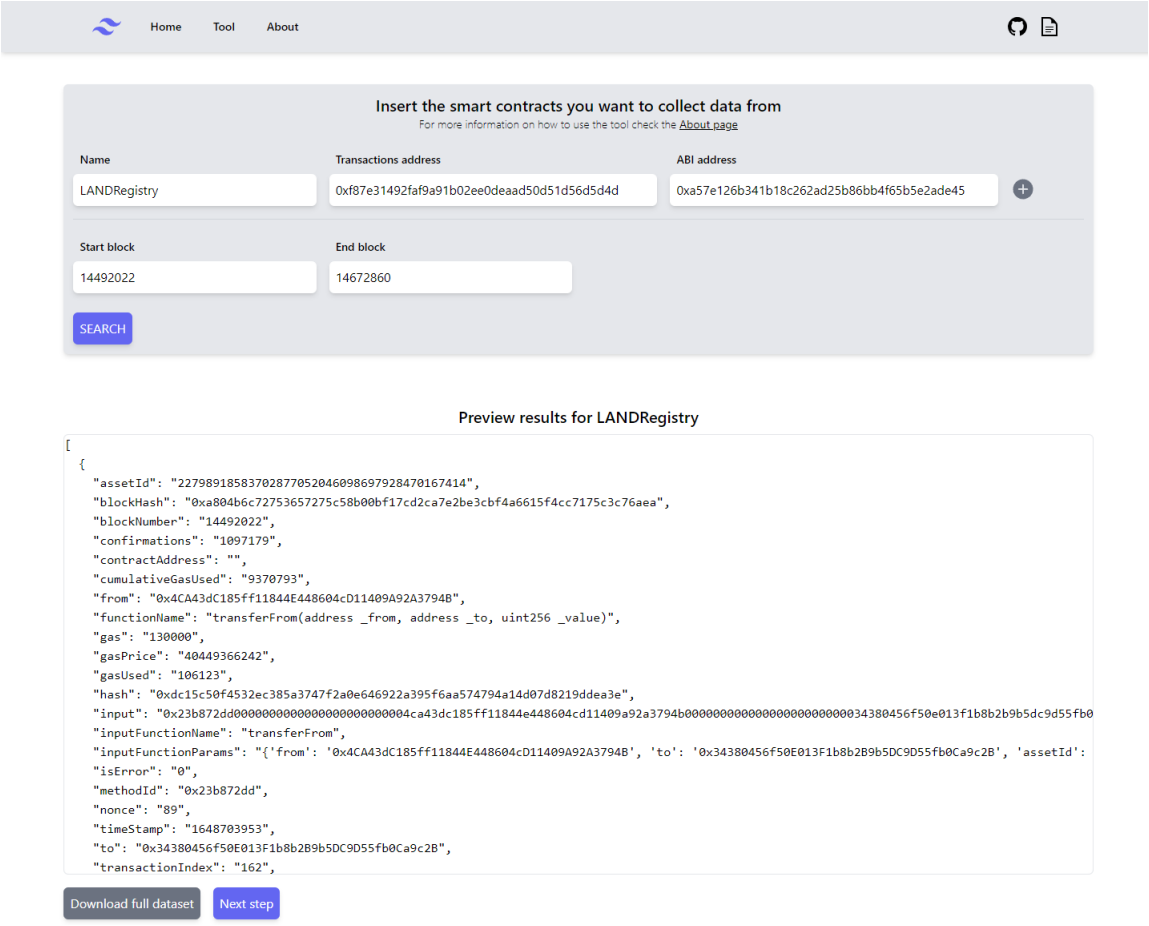

Figure 4.4: Transactions collection page

The second step starts with the transactions fetched earlier. As depicted in figure 4.5, the smart contracts and block range inserted in the previous phase are listed to highlight the subjects of the operation. The parameters required here are:

- *Sort by*: the parameter by which the sorting of the records is done.
- *Case ID*: the parameter selected to build traces. For instance, selecting *"from"* will lead to a trace for each user that has interacted with the contract.

- *Concept name*: the *case:concept:name* value in the XES event. In general, it is used in process discovery as the name of the activities contained in the output model.

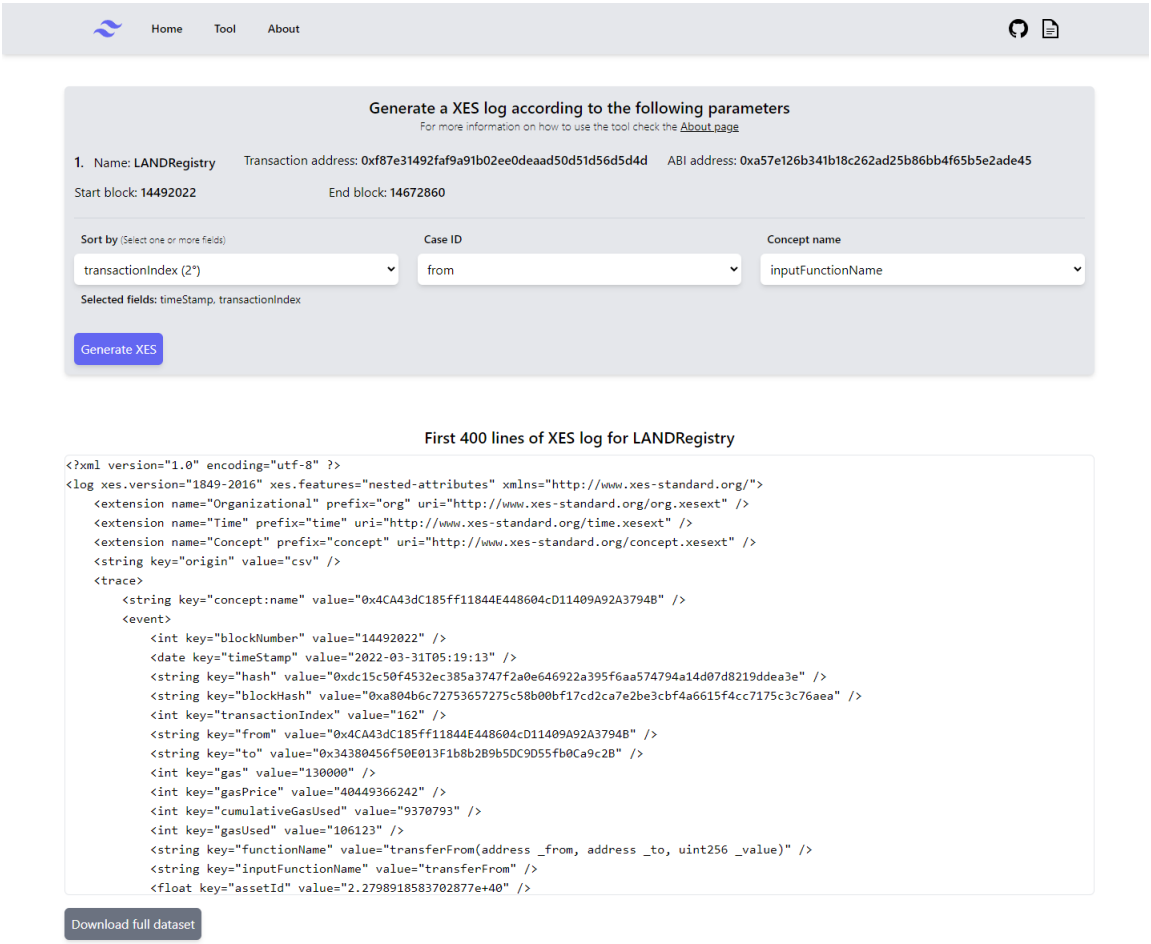

Figure 4.5: XES generation page

These parameters can be selected through a dropdown containing the standard Ethereum transactions fields plus the parameters extracted from the smart contract function executed in the transactions. For instance, from the function *updateLandData(int256 x, int256 y, string data)* the framework extracted *x*, *y*, and *data*. Clicking on the *"Generate XES"* button will show a preview of the generated XES log and a button to download the full XES log. The generated log can be successfully used as input for every process mining technique accepting XES logs.

5. Use case

To demonstrate the validity of the proposed methodology, it was executed on Decentraland, a metaverse and digital asset marketplace developed on the Ethereum blockchain. It was possible to generate XES logs from different ranges of blocks and one or more smart contracts combined. Process mining techniques were successfully performed on the generated logs. The results were evaluated through conformance checking algorithms considering fitness, precision, generalization, and simplicity. The following sections detail the application of the methodology.

5.1 Decentraland

Decentraland [17] is a metaverse and digital assets marketplace developed on the Ethereum blockchain. Within Decentraland, users can create, experience, and monetize content and applications. The finite, traversable, 3D virtual space within Decentraland is called LAND, a non-fungible digital asset maintained in an Ethereum smart contract. The land is divided into parcels that are identified by cartesian coordinates (x,y). These parcels are permanently owned by members of the community and are purchased using MANA, Decentraland's cryptocurrency token. This gives users full control over the environments and applications they create, which can range from anything like static 3D scenes to more interactive applications or games. Some parcels are further organized into themed communities or Districts. By organizing parcels into Districts or ESTATEs, the community can create shared spaces with common interests and uses. Unlike other virtual worlds and social networks, Decentraland is not controlled by a centralized organization. There is no single agent with the power to modify the rules of the software, contents of land, and economics of the currency, or prevent others from accessing the world.

Figure 5.1: Decentraland logo

Speaking more technically, Decentraland has four main smart contracts:

- MANA[1]. It is the ERC20[2] smart contract that manages the MANA token. MANA is the Decentraland's cryptocurrency which is used to buy LANDs, ESTATEs, and

[1] https://etherscan.io/token/0x0f5d2fb29fb7d3cfee444a200298f468908cc942
[2] https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

any other collectibles of the ecosystem. It is also used to vote for proposals in the Decentraland DAO (i.e., "Decentralized Autonomous Organization").

- LANDRegistry[3]. It is a registry that tracks the ownership of each LAND and the attached information: owner, metadata, version, name, description, and ipns. It provides functions to transfers parcels (i.e., *transferFrom*, *transferLand*), create ESTATEs (i.e., *createEstate*), updated parcels metadata (i.e., *updateLandData*), and others.

- ESTATERegistry[4]. It is similar to LANDRegistry but tracks the ownership of ESTATEs which are an association of two or more directly adjacent parcels of LAND. It contains functions similar to LANDRegistry but for ESTATEs.

- Marketplace[5]. The Marketplace contract allows every LAND and ESTATE in Decentraland to be exchanged for MANA by a smart contract which provides the ability to do a trustless and transparent atomic swap. Within the marketplace, users will be able to use their MANA to buy ERC721[6] tokens (not only parcels are sold in Decentraland but also avatars, gadgets, and more). Every order (i.e. NFT listing) has its unique id, NFT owner, and price. It provides functions to list items (i.e., *createOrder*), execute orders (i.e., *executeOrder*), cancel orders (i.e., *cancelOrder*), and others.

Decentraland was chosen as a use case for three reasons:

- Decentraland provides several analyzable features: cryptocurrencies and NFTs exchange, virtual lands, games, and more. A large number of features means a large number of analyzable processes, especially with process mining.

- High number of available transactions. The high number of functions and smart contracts result in a large number of analyzable transactions.

- To our knowledge, process mining hasn't been extensively applied to metaverses yet. Metaverses are an emergent and exciting sector yet to be explored by techniques like process mining.

5.2 Methodology application

The following sections describe the execution of the methodology with the three different inputs (i.e., transactions, internal transactions, and the combination). The methodology was tested on the four smart contracts mentioned earlier. As an example, in the following section, the execution of LANDRegistry is described since it is the most utilized smart contract in Decentraland. LANDRegistry was developed using the proxy pattern, so two smart contract addresses are required, LANDProxy[7] to collect transactions, and LANDRegistry[8] to get the ABI.

[3] `https://etherscan.io/address/0xa57e126b341b18c262ad25b86bb4f65b5e2ade45`
[4] `https://etherscan.io/address/0x1784ef41af86e97f8d28afe95b573a24aeda966e`
[5] `https://etherscan.io/address/0x19a8ed4860007a66805782ed7e0bed4e44fc6717`
[6] `https://ethereum.org/en/developers/docs/standards/tokens/erc-721/`
[7] `https://etherscan.io/address/0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d`
[8] `https://etherscan.io/address/0xa57e126b341b18c262ad25b86bb4f65b5e2ade45`

5.2.1 Transactions

The first execution was conducted on transactions. LANDProxy was selected as the transaction address and LANDRegistry as the ABI address due to the proxy pattern. Regarding the block range, three different combinations were tried to test the methodology performance on logs with different sizes:

- *14492022 - 14672860*: small-size log with 540 traces and 1000 events. Related to transactions from *31 March 2022* to *28 April 2022*.
- *12322265 - 14883354*: intermediate-size log with 4895 traces and 12299 events. Related to transactions from *27 April 2021* to *01 June 2022*.
- *0 - 14590157*: full-size log with 9184 traces and 51855 events. Related to transactions from *21 January 2018* (i.e., LANDRegistry deploy date) to *15 April 2022* (i.e., transactions collection date).

For all the specified block ranges, *timestamp* was utilized for sorting the records, *from* (i.e., sender) was selected as *case ID*, and *inputFunctionName* as *concept:name*. Selecting the sender as case ID groups into traces all the transactions sent by a user towards LANDProxy. It means that a trace represents the lifecycle of transactions sent by an address. An example trace from the generated log is shown in figure 5.2. The *concept:name* (i.e. case ID) is equal to the *from* attribute in the events belonging to the trace. The events have the *concept:name* attribute equal to the *inputFunctionName* attribute.


```
<trace>
    <string key="concept:name" value="0x0688d9c02df342da756b8acfd135e3761db3915c" />
    <event>
        <int key="blockNumber" value="12322265" />
        <date key="timeStamp" value="2021-04-27T11:56:19" />
        <string key="hash" value="0x492c87de3bcb6a632d2a0f5cb4f205c0cf8518a565b0e88ba85bf56990e807ca" />
        <string key="blockHash" value="0xdae0471135c4319068952d7f85f4cb3e75cae09b8f054290ccbda39b7ba8b12b" />
        <int key="transactionIndex" value="18" />
        <string key="from" value="0x0688d9c02df342da756b8acfd135e3761db3915c" />
        <string key="to" value="0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d" />
        <int key="value" value="0" />
        <string key="input" value="0xd4dd159400000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000064ffffffffffffff
        <string key="inputFunction" value="&lt;Function updateLandData(int256,int256,string)&gt;" />
        <string key="inputFunctionName" value="updateLandData" />
        <string key="inputFunctionParams" value="{'x': 100, 'y': -149, 'data': '0,&quot;Libertopia&quot;,&quot;&quot;,'}" />
        <date key="TIMESTAMP" value="2021-04-27T11:56:20" />
        <string key="org:resource" value="0x0688d9c02df342da756b8acfd135e3761db3915c" />
        <date key="time:timestamp" value="2021-04-27T11:56:19" />
        <string key="concept:name" value="updateLandData" />
    </event>
    <event>
        <int key="blockNumber" value="12554630" />
        <date key="timeStamp" value="2021-06-02T10:53:47" />
        <string key="hash" value="0x4ef6a430db6a57fc13f1a3c4da70f28f7004ebc6fed16362f0d61f1ca5df3d4f" />
        <string key="blockHash" value="0x4a8301ff41b582b0e9b620bde027e26b2398d45635688bcbd0f180448b61707d" />
        <int key="transactionIndex" value="179" />
        <string key="from" value="0x0688d9c02df342da756b8acfd135e3761db3915c" />
        <string key="to" value="0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d" />
        <int key="value" value="0" />
        <string key="input" value="0xd4dd159400000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000064ffffffffffffff
        <string key="inputFunction" value="&lt;Function updateLandData(int256,int256,string)&gt;" />
        <string key="inputFunctionName" value="updateLandData" />
        <string key="inputFunctionParams" value="{'x': 100, 'y': -149, 'data': '0,&quot;Flotopia&quot;,&quot;&quot;,'}" />
        <date key="TIMESTAMP" value="2021-06-02T10:53:48" />
        <string key="org:resource" value="0x0688d9c02df342da756b8acfd135e3761db3915c" />
        <date key="time:timestamp" value="2021-06-02T10:53:47" />
        <string key="concept:name" value="updateLandData" />
    </event>
</trace>
```


Figure 5.2: XES trace generated from transactions

5.2.2 Internal transactions

In the collection of internal transactions for LANDProxy for the maximum block range, *600.000* records were obtained. As depicted in Chapter 3, the application of the methodology should generate a log for each function name discovered in the dataset: 33 logs were generated. Each log contains the internal transactions for the transactions executing a specific function. For instance, one log for the function *setApprovalForAll*, another for *createEstate*, *transferFrom*, and so on. The *timestamp* was used to sort the records, the *hash* of the parent transaction was selected as case ID, and the *function name* as *concept:name*. In this manner, each trace represents the internal execution flow of a transaction. An example trace from the generated logs is shown in figure 5.3. The *concept:name* (i.e. case ID) is equal to the *TX_HASH* attribute in the events belonging to the trace. The events have the *concept:name* attribute equal to the *FUNCTION_NAME* attribute that was prefixed with the value of the *FROM_NAME* attribute.

The execution of techniques like process discovery on such logs should produce models that resemble the execution flow with splits in correspondence of decisions (e.g., *if, else*) and loops in correspondence of *for* or *while* constructs. Infrequent variants or unusual branches in the model could represent bugs in the code.


```
<trace>
    <string key="concept:name" value="0x1cb3e579f25d684aef00b8bb52871ba7eac6f787f120cbabfca3df192d038e9a" />
    <event>
        <int key="BLOCK" value="7109761" />
        <date key="TIMESTAMP" value="2020-05-25T02:37:36" />
        <string key="TX_HASH" value="0x1cb3e579f25d684aef00b8bb52871ba7eac6f787f120cbabfca3df192d038e9a" />
        <string key="CALL_ID" value="0" />
        <string key="CALL_TYPE" value="delegatecall" />
        <string key="FROM_ADDRESS" value="0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d" />
        <string key="FROM_NAME" value="LAND" />
        <string key="TO_ADDRESS" value="0xf8550b0511486af9a75c63d85a01c919535f76db" />
        <string key="TO_NAME" value="LANDRegistry" />
        <string key="FUNCTION_SIGNATURE" value="0xca8a2c08" />
        <string key="FUNCTION_NAME" value="LAND_createEstate" />
        <string key="ARGUMENTS" value='{"beneficiary":"0x5b50fb8281cb679a3102d3de22dd05a1ad895e8f","x":[150],"y":[9]}' />
        <string key="OUTPUTS" value='{"__out0":1954}' />
        <int key="ORDER_INDEX" value="3160" />
        <string key="ORIGIN_ADDRESS" value="0xbe3fbebc1cc29e99e02237dac84860e268cabc9b" />
        <string key="org:resource" value="0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d" />
        <date key="time:timestamp" value="2020-05-25T02:37:36" />
        <string key="concept:name" value="LAND_createEstate" />
    </event>
    <event>
        <int key="BLOCK" value="7109761" />
        <date key="TIMESTAMP" value="2020-05-25T02:37:31" />
        <string key="TX_HASH" value="0x1cb3e579f25d684aef00b8bb52871ba7eac6f787f120cbabfca3df192d038e9a" />
        <string key="CALL_ID" value="0_0" />
        <string key="CALL_TYPE" value="call" />
        <string key="FROM_ADDRESS" value="0xf8550b0511486af9a75c63d85a01c919535f76db" />
        <string key="FROM_NAME" value="LANDRegistry" />
        <string key="TO_ADDRESS" value="0x959e104e1a4db6317fa58f8295f586e1a978c297" />
        <string key="TO_NAME" value="EstateRegistry" />
        <string key="FUNCTION_SIGNATURE" value="0xd0def521" />
        <string key="FUNCTION_NAME" value="LANDRegistry_mint" />
        <string key="ARGUMENTS" value='{"metadata":"","to":"0x5b50fb8281cb679a3102d3de22dd05a1ad895e8f"}' />
        <string key="OUTPUTS" value='{"__out0":1954}' />
        <int key="ORDER_INDEX" value="3162" />
        <string key="ORIGIN_ADDRESS" value="0xbe3fbebc1cc29e99e02237dac84860e268cabc9b" />
        <string key="org:resource" value="0xf8550b0511486af9a75c63d85a01c919535f76db" />
        <date key="time:timestamp" value="2020-05-25T02:37:31" />
        <string key="concept:name" value="LANDRegistry_mint" />
    </event>
</trace>
```


Figure 5.3: XES trace generated from internal transactions

5.2.3 Combination of transactions and internal transactions

A subset of the dataset with 600.000 transactions was selected to process the combination of transactions and their internal transactions. The choice was due to the need to avoid performance issues in the execution of process mining techniques. The *timestamp* was used to sort the records, the *from* the field was selected as case ID, and the *function name* as *concept:name*. As described in Chapter 3, each function name is prefixed by the smart contract name on parent transactions (e.g., *LAND.transfer*) and by {*function_name*}_{*from*} on internal transactions (e.g., *transfer_LAND.transfer*). An example trace from the generated log is shown in figure 5.4. The *concept:name* (i.e. case ID) is equal to the *id* attribute in the events belonging to the trace. The *concept:name* of the events representing parent-transactions is obtained by prefixing the smart contract name to the function name (e.g., *LAND.transfer*). Instead, the *concept:name* of events representing internal transactions is obtained by prefixing also the function name of the parent transaction (e.g., *transfer_LAND.transfer*). The internal transactions events contain the attribute with *FK_* as the prefix to indicate that they're internal transactions during the BPMN Miner execution.

```
<trace>
    <string key="concept:name" value="0x055f66a745ee38cc76c9f5e68c4c6968f14761da" />
    <event>
        <string key="id" value="0x055f66a745ee38cc76c9f5e68c4c6968f14761da" />
        <date key="time:timestamp" value="2018-01-23T08:37:27+00:00" />
        <string key="concept:name" value="LAND.transfer" />
    </event>
    <event>
        <string key="id" value="0x055f66a745ee38cc76c9f5e68c4c6968f14761da" />
        <string key="transfer" value="0x71e555ab18640bf632546d397cad071d2b676d0893ebc7ce5641925d9279dd38" />
        <date key="time:timestamp" value="2018-01-23T08:37:27+00:00" />
        <string key="concept:name" value="transfer_LAND.transfer" />
    </event>
    <event>
        <string key="id" value="0x055f66a745ee38cc76c9f5e68c4c6968f14761da" />
        <date key="time:timestamp" value="2018-01-30T06:20:55+00:00" />
        <string key="concept:name" value="LANDRegistry.transferManyLand" />
    </event>
    <event>
        <string key="id" value="0x055f66a745ee38cc76c9f5e68c4c6968f14761da" />
        <string key="FK_transferManyLand" value="0x8b02a79315efc57558c35b1b2aa28173c2359ec7f1c89a4717b4cc076e7fb9f6" />
        <date key="time:timestamp" value="2018-01-30T06:20:55+00:00" />
        <string key="concept:name" value="transferManyLand_LAND.transferManyLand" />
    </event>
</trace>
```

Figure 5.4: XES trace generated for BPMN Miner

5.3 Process mining

Process mining techniques, such as process discovery and conformance checking, were applied to the generated XES logs. Compliance checking was performed on the discovered model to evaluate the performance of the discovery algorithms on the logs. Special attention was given to the logs generated by combining transactions and internal transactions, on which BPMN Miner [7], a technique capable of producing a BPMN model with sub-processes, boundary events, and activity markers, was applied.

5.3.1 Statistics

Before performing the process mining techniques, some statistics were calculated using PM4Py to get a better understanding of the logs. The first is the distribution of events over time, which helps to understand in which time intervals the most events are recorded. It can be shown as a graph or plot over different time intervals: hours, weeks, days, months, and years. The figure 5.5 shows the events distribution plot for the full log. The peak of events occurred in the first few months after the launch of Decentraland and the last months of 2021, when Facebook decided to rebrand itself as Meta [9], putting a massive focus on metaverses. In terms of smaller timeframes, during the winter months and the evening hours, the volume of transactions is higher.

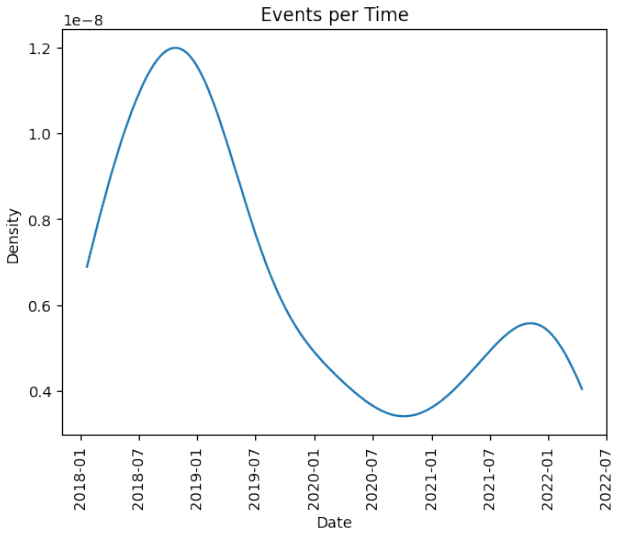

Figure 5.5: Events distribution plot for the full log

Another interesting chart is the dotted chart. It is a classic visualization of the events inside an event log across different dimensions (X, Y, colors) where each event of the event log corresponds to a point. The dotted chart can be built by selecting different attributes. A convenient choice is to display the distribution of cases and events over time, with the following: X-axis for event timestamp, Y-axis for case index within the event log, and Color for event activity. The aforementioned choices permit the identification of visual patterns, e.g., vertical patterns and horizontal patterns. The formers indicate shorter periods where the same activity occurs in many cases. Vertical patterns can be seen in the correspondence of the chart being more inclined. Also, horizontal patterns are interesting: since the Y-axis represents the traces (i.e., users), a horizontal pattern indicates a specific activity being executed frequently. In figure 5.6, vertical patterns can be seen at the beginning and in correspondence of late 2021 (i.e., the end of the chart). Instead, horizontal patterns can be seen in the lower part of the chart. In general, most of the dots in the chart represent the functions*setApprovalForAll*, *transferFrom*, and *updateLandData*. It indicates that these three functions are the most executed ones.

The filtering on variants shows that the most frequent trace is, by far, composed just by *setApprovalForAll*. This function specifies an operator which will be able to operate on the LAND on behalf of the address calling the function. In many cases, the address

[9] `https://about.fb.com/news/2021/10/facebook-company-is-now-meta/`

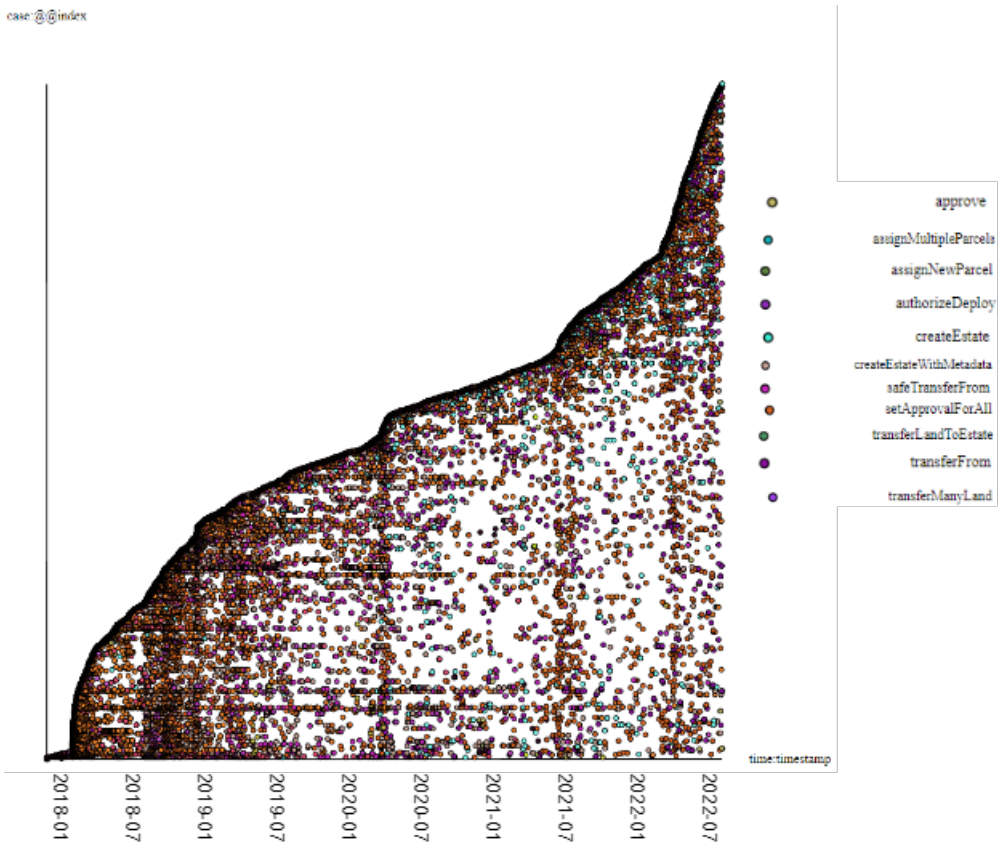

Figure 5.6: Dotted chart for the full log

of the operator refers to the Marketplace contract meaning that *setApprovalForAll* was executed to let the latter sell the LAND. It explains why it is also the function that occurs the most as the first activity in a trace.

The statistics described show that XES logs can be used for other purposes besides process mining and that the proposed extraction methodology can similarly serve multiple intents.

5.3.2 Process discovery and Conformance checking

The first process mining technique applied to the generated logs was process discovery. PM4Py was used to execute the following discovery algorithms:

- **Inductive Miner (IM)** [11], **Inductive Miner infrequent (IMf)** [12], and **Inductive Miner directly-follows (IMd)** [13]. The basic idea of Inductive Miner is about detecting a "cut" in the log (e.g. sequential cut, parallel cut, concurrent cut, and loop cut) and then recur on sub-logs, which were found applying the cut until a base case is found. Inductive miner models usually make extensive use of hidden transitions, especially for skipping/looping on a portion of the model. Furthermore, each visible transition has a unique label (there are no transitions in the model that share the same label). The base variant (IM) produces models with perfect replay fitness. The infrequent variant (IMf) produces a more precise model, without a fitness guarantee, by eliminating some behavior. The directly-follows variant (IMd) avoids the recursion on the sub-logs but uses

the directly follows graph. Maximum performance but replay fitness guarantees are lost. In figures 5.7, 5.8, 5.9 are respectively shown the models discovered on the one-year log with Inductive Miner directly-follows (IMd), Inductive Miner infrequent (IMf), and Inductive Miner (IM). The results on the full log are similar, the only difference being the presence of the deploy and setup functions executed when the smart contracts were deployed. The model discovered with IM (i.e., figure 5.9) and IMd (i.e., figure 5.7 are pretty similar. Both the models have a big loop including all the activities except *setManyUpdateOperator*. The only difference is the upper part because in the IM model there are more decisions between *transferManyLandToEstate* and *transferLandToEstate*. Instead, the model discovered with IMf (i.e., figure 5.8) contains more decisions and is more structured.

- **Heuristic Miner** [21]. It is an algorithm that acts on the directly-follows graph, providing a way to handle noise and find common constructs (e.g., a dependency between two activities, AND). The output of the Heuristics Miner is a Heuristics net, an object that contains the activities and the relationships between them. The Heuristic Miner was executed with default parameters: dependency_threshold 0.5, and_threshold 0.65, and loop_two_threshold (0.5). The model discovered with the Heuristic Miner is shown in Figure 5.10. It was executed with default parameters since the goal of this thesis is to demonstrate the validity of the extraction methodology and not obtain the best model.

- **Alpha algorithm** [1]. The Alpha algorithm is one of the most known process discovery algorithms and is able to find: a Petri net model where all the transitions are visible and unique and correspond to classified events (for example, to activities); an initial marking that describes the status of the Petri net model when execution starts; a final marking that describes the status of the Petri net model when execution ends.

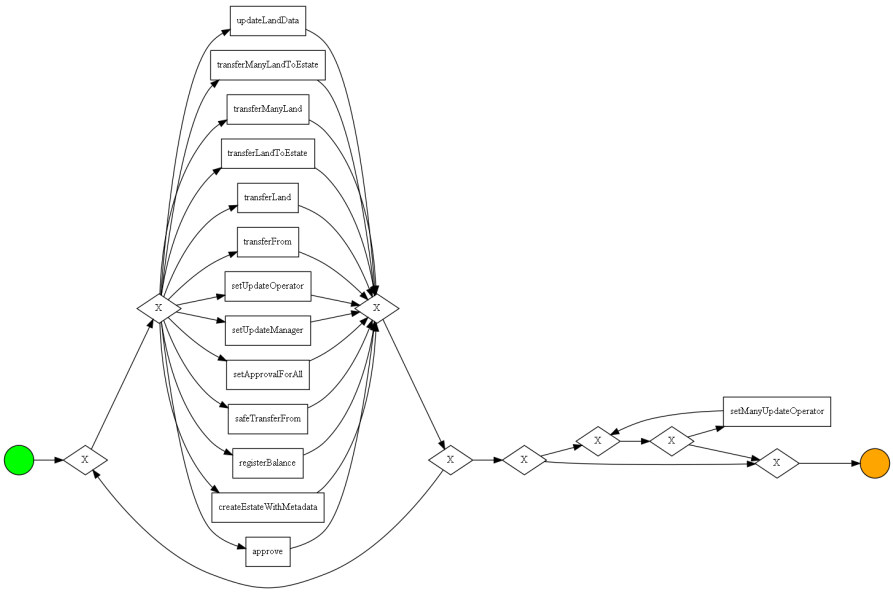

Figure 5.7: IMd model on one-year log

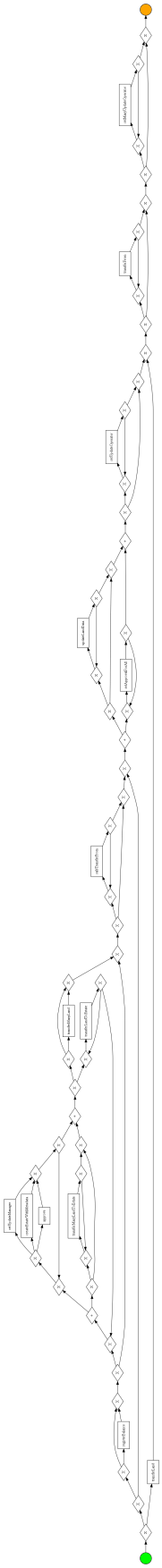

Figure 5.8: IMf model on one-year log

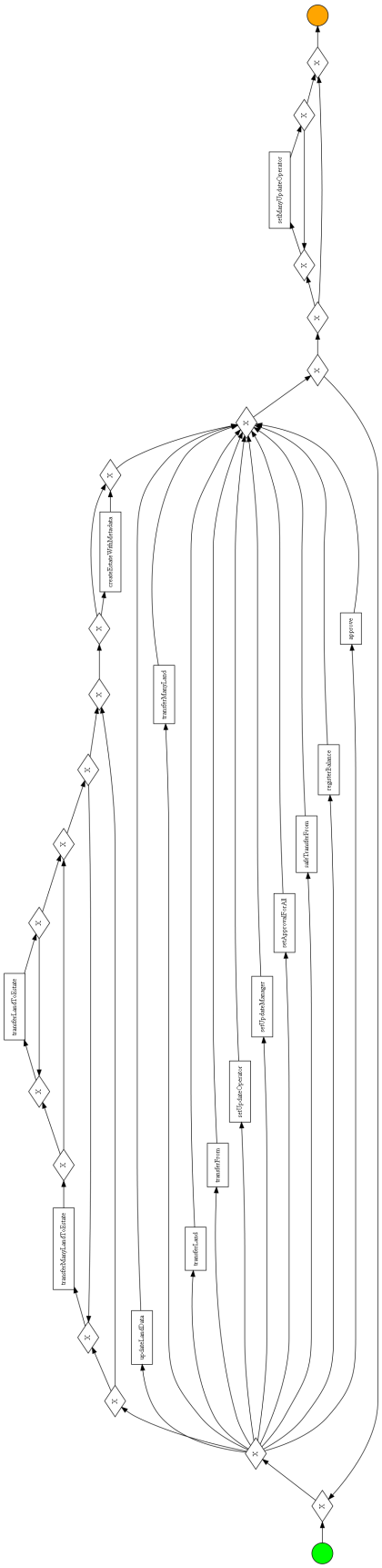

Figure 5.9: IM model on one-year log

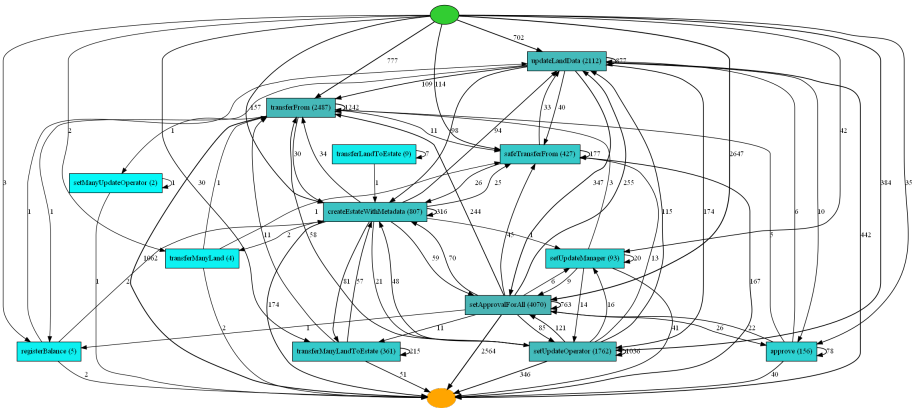

Figure 5.10: Heuristic miner model on one-year log

Overall the results don't bring any additional knowledge. However, this is more related to the nature of the analyzed smart contracts than the methodology and the applied process discovery algorithms. In Decentraland, users buy a LAND, and then they can perform different actions: sell the LAND for a profit/loss, deploy games on the LAND, add decorations, add the LAND to an ESTATE, and more. Such behaviors are not structured and don't have any causal dependencies between them. That's why there isn't a clear structure in the model. The only recurrent behavior is *setManyUpdateOperator* as the last activity since it is executed to set the Marketplace as the LAND operator and sell it. Extracting logs from more Decentraland smart contracts combined (e.g., LANDRegistry, Marketplace) will result in a more structured model and clear causal dependencies (e.g., *buy LAND* as the first activity).

The metrics provided by PM4Py[10] were employed to assess the discovered models. Such metrics can be obtained by executing conformance checking techniques. The following are the considered metrics:

- **Fitness**. Aim to calculate how much of the behavior in the log is admitted by the process model. It supports both token-based replay and alignment-based replay. For token-based, the percentage of traces that are completely fit is returned along with a fitness value that is calculated as indicated in [3]. For alignments, the percentage of traces that are completely fit is returned, along with a fitness value that is calculated as the average of the fitness values of the single traces. The token-based replay was chosen because it performed better on bigger logs.

- **Precision**. Two approaches for the measurement of precision in PM4Py, *ET-Conformance* [16] which uses token-based replay and *Align-ETConformance*[16] which uses alignments. The idea underlying the two approaches is the same: the different prefixes of the log are replayed (whether possible) on the model. At the reached marking, the set of transitions that are enabled in the process model is compared with the set of activities that follow the prefix. The more the sets are different, the more the precision value is low. The more the sets are similar, the more the precision value is high. This works only if the replay of the prefix on the process model works: if the replay does not produce a result, the prefix is not considered for the computation of precision. Hence, the precision calculated on top of unfit processes is not meaningful. The main difference between the approaches

[10] `https://pm4py.fit.fraunhofer.de/documentation#evaluation`

is the replay method. Token-based replay is faster but based on heuristics (hence the result of the replay might not be exact). Alignments are exact, and work on any kind of relaxed sound nets, but can be slow if the state-space is huge. Also for precision, the token-based replay was chosen because it performed better on bigger logs.

- **Generalization**. Analyze how the log and the process model match. PM4Py generalization measure is described in [5]. A model is general whether the elements of the model are visited enough often during a replay operation (of the log on the model). A model may be perfectly fitting the log and perfectly precise (e.g., reporting the traces of the log as sequential models going from the initial marking to the final marking; a choice is operated at the initial marking). Hence, to measure generalization a token-based replay operation is performed, and the generalization is calculated as $1 - avg_t(sqrt(1.0/freq(t)))$ where avg_t is the average of the inner value over all the transitions, $sqrt$ is the square root, $freq(t)$ is the frequency of t after the replay.

- **Simplicity**. PM4Py defines simplicity by taking into account only the Petri net model. The criteria chosen for simplicity is the inverse arc degree as described in [4]. First of all, the average degree for a place/transition of the Petri net is considered. It is defined as the sum of the number of input arcs and output arcs. If all the places have at least one input arc and output arc, the number is at least 2. Choosing a number k between 0 and infinity, the simplicity based on the inverse arc degree is then defined as $1.0/(1.0 + max(mean_degree - k, 0))$.

The table 5.1 lists the results of the conformance checking step. The Inductive Miner infrequent (IMf) is the process discovery algorithm that performed better, followed by the Heuristic Miner and the Inductive Miner directly-follows. The metrics and the model discovered by the Heuristic Miner could be improved by tuning parameters like *dependency threshold* and *noise threshold*. The arcs with low frequency present in figure 5.10 will be removed resulting in a clearer model. In general, the properties of the Inductive Miner variants are reflected in the models. The IMd model in figure 5.7 is the most understandable but lacks precision. Instead, the IMf produces a model with high fitness, precision, and generalization. The Alpha algorithm is the discovery algorithm that performed the worst. However, it was predictable since it was the first conceptualized algorithm with limits and issues that were overcome by the newest algorithm.

The same execution of transactions has been done on the XES logs generated from internal transactions. The same process discovery algorithms were applied and metrics similar to table 5.1 were obtained. The figure 5.11 shows the model discovered with the Inductive Miner infrequent (IMf) on the XES log of the *transferLandToEstate* function. The model represents the execution flow of the function with loops and OR-gateways in correspondence of, respectively, iterations (e.g., for, while) and decisions (e.g., if, else, switch).

5.3.3 BPMN Miner

The second application of process mining was the BPMN Miner [7] algorithm. It is a technique able to output a BPMN model with sub-processes, boundary events, and activity markers inside. As explained in Chapter 3, the conditions behind BPMN Miner

Algorithm	**Fitness**[3]	**Precision**[16]	**Generalization**[5]	**Simplicity**[4]
IM[11]	1	0,56	0,77	0,62
IMf[12]	0,96	0,93	0,84	0,7
IMd[13]	1	0,53	0,77	0,55
Heuristic[21]	0,98	0,84	0,7	0,46
Alpha[1]	0,56	0,8	0,8	0,47

Table 5.1: Process discovery metrics for transactions

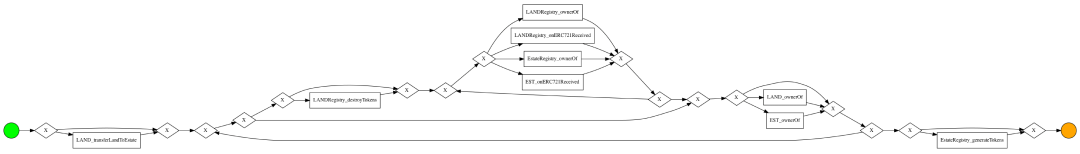

Figure 5.11: Model discovered from transferLandToEstate with the IMf

are similar to the notions of the key and foreign key in relational databases. The algorithm seeks to split the log into sub-logs based on process instance identifiers (i.e., keys) and references from sub-process to parent process instances (i.e., foreign keys). The process and sub-process relations create a hierarchy that is used to generate a BPMN model with sub-processes, boundary events, and activity markers. The model generated with BPMN Miner will contain the same activities as the models generated using transaction data (e.g., figure 5.7) but with some activities expanded to include the sub-process. Moreover, the sub-processes are the ones shown in the models generated with internal transactions (e.g., 5.11. Basically, the model discovered with BPMN Miner is a combination of the models discovered with transactions and internal transactions.

As noted earlier, a subset of the 600.000 records dataset was selected to generate an XES log with 1604 traces, 25249 events, and 17 different event types. BPMN Miner is available as a plugin in ProM, specifically in nightly builds[11]. BPMN Miner works on top of existing process discovery algorithms, it supports the Inductive Miner, the Heuristic Miner, the ILP Miner[22], and the Alpha algorithm. The Inductive Miner, the Heuristic Miner, and the Alpha algorithm were selected to be coherent with the execution performed on transactions and internal transactions. BPMN Miner also permits the tuning of parameters such as noise thresholds and tolerance values, but the default values displayed in figure 5.12 were used.

The next step is the selection of the attributes as primary keys to creating the relations and, subsequently, the sub-processes. In Chapter 3, it was mentioned that the attributes representing functions without internal transactions were prefixed with *FK*. As depicted in figure 5.13, such attributes should not be selected as primary keys since they don't lead to sub-processes. In this step, the algorithm starts to create the hierarchy of primary keys and foreign keys.

[11] `https://www.promtools.org/doku.php?id=nightly`

Figure 5.12: Algorithm selection step of BPMN Miner on ProM

Some event types may be added to the hierarchy of multiple primary keys. BPMN Miner allows selecting which of the available keys the events types should be a foreign key. In order to ease the selection, as explained in Chapter 3, event types were prefixed with the *function_name* of the parent transactions they belong. For instance, in figure 5.14, *transferFrom* should be selected as primary key for *transferFrom_LANDRegistry.destroyTokens*.

Once the selection of the primary key is complete, BPMN Miner is ready to generate the output model. An example is shown in Figure 5.15. As can be seen, the model has a similar structure to that generated with Inductive Miner infrequent (IMf) on logs extracted from transactions. The difference is that some activities, e.g., *registerBalance*, *transferFrom*, *createEstateWithMetadata*, are expanded to contain their sub-process.

To evaluate the performance of the BPMN Miner algorithm, the same metrics and process used to calculate transactions and internal transactions were applied (i.e., fitness, accuracy, generalization, and simplicity). The table 5.2 shows the results. Also, in this case, the Inductive Miner is the best-performing algorithm.

Algorithm	**Fitness**[3]	**Precision**[16]	**Generalization**[5]	**Simplicity**[4]
$\text{BPMNMiner}_{\text{Inductive}}$	0,84	0,67	0,73	0,65
$\text{BPMNMiner}_{\text{Heuristic}}$	0,96	0,69	0,54	0,53
$\text{BPMNMiner}_{\text{Alpha}}$	0,77	0,67	0,76	0,18

Table 5.2: BPMN Miner metrics

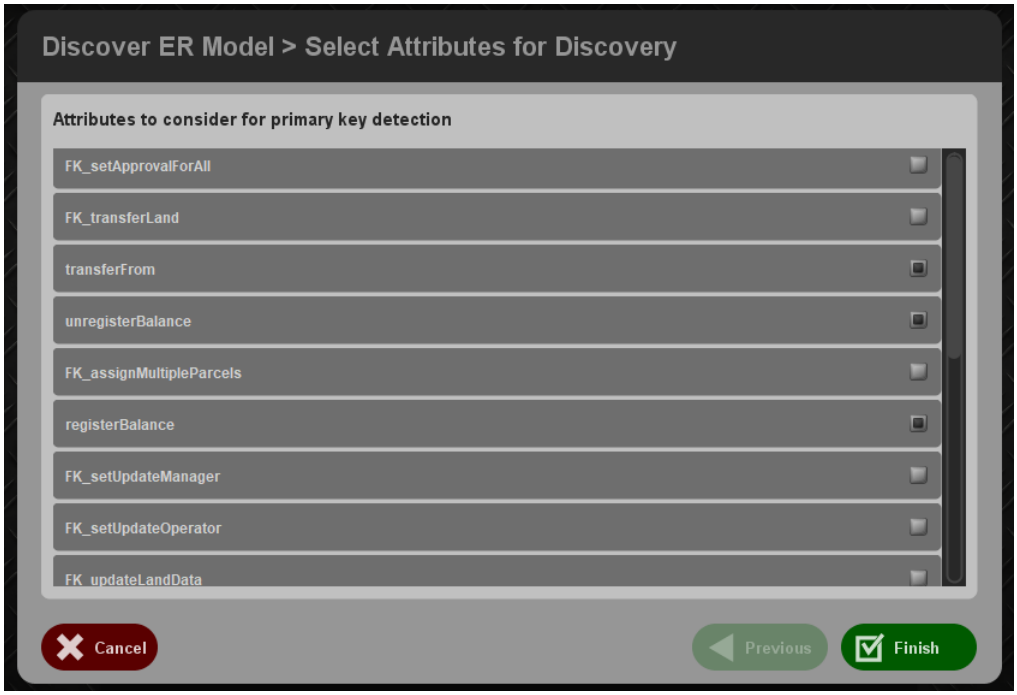

Figure 5.13: Attributes selection step of BPMN Miner on ProM

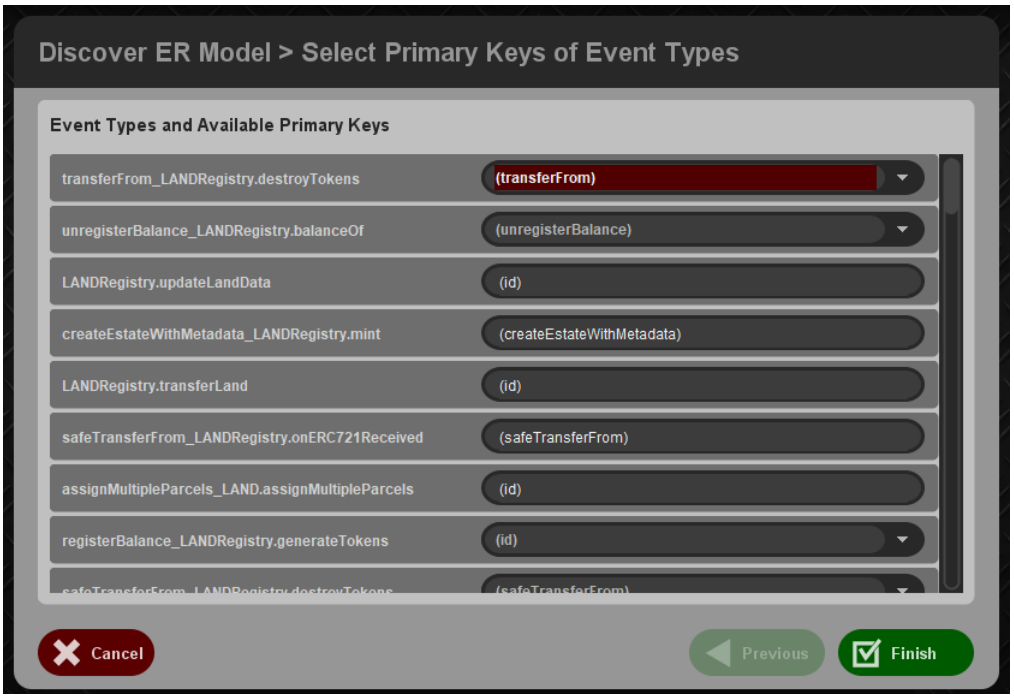

Figure 5.14: Primary key selection step of BPMN Miner on ProM

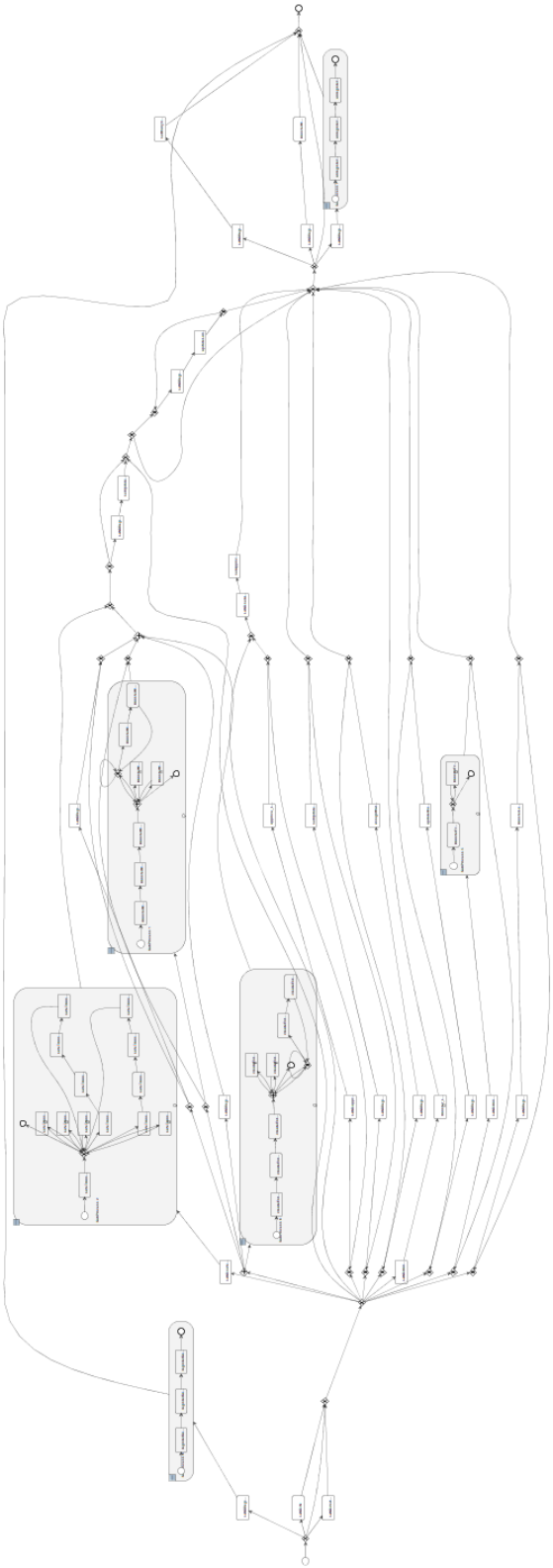

Figure 5.15: Model discovered with $\text{BPMNMiner}_{\text{Inductive}}$

6. Related work

Before beginning the research, a review of existing work on process mining and blockchain was conducted. The text parameter *"process mining blockchain"* was used on *Google Scholar* to search for contributions made in recent years by researchers on the topic. To skim the results obtained, only articles that met the following criteria were analyzed:

- Describes the implementation of a log-extraction methodology.
- Deals with the Ethereum blockchain or EVM-compatible blockchains.
- Has a considerable number of citations (i.e., >50)

After the filtering, papers dealing with not EVM-compatible blockchains such as [2] and [8] were not further analyzed. The remaining papers were: *"Extracting Event Logs for Process Mining from Data Stored on the Blockchain"*[15], *"Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications"*[10], and *"Process Mining on Blockchain Data: A Case Study of Augur"*[19]. The examination of the mentioned works highlighted limitations and room for additional research. The table 6.1 presents an overview of the analyzed papers with the *methodology*, the *target data*, the *target application*, *case ID*, and the *process discovery algorithm* since it was applied in all the three papers.

Methodology	**Data**	**Target**	**Case ID**	**Discovery algorithm**
ELF[10]	Event	CryptoKitties	Kitty ID	DFGs
ELF[19]	Event	Augur	Market ID	Inductive Miner
BPMS[15]	Transaction	Incident-management[20]	transaction.to	Inductive Visual Miner

Table 6.1: Related work overview

The following sections will describe the three papers and their limitations.

6.1 Process mining on blockchain-based BPMS

Di Ciccio et al. in [15] defined a methodology to extract logs from the transactions involving smart contracts generated with a blockchain-based Business Process Management System (BPMS) like Caterpillar[14] and Lorikeet[18]. This kind of system is able to translate BPMN or Choreography models into smart contracts. The generated smart

contracts resemble the execution flow and constraints represented in the input models. Caterpillar and Lorikeet use a factory contract to deploy each new smart contract, i.e., process instance. The address of the factory contract is included in the *"to"* field of the blockchain transaction associated with the creation of a new process instance. If the factory contract address is provided, the address of each process instance deployed is identifiable. Moreover, other information is known a priori: (i) the context and the scenario where the operations take place, (ii) the set of allowed activities, (iii) and the execution flow of the system. These assumptions do not apply to most of the smart contracts deployed in the Ethereum blockchain because, even if they were present, it would not be possible to have access to the models underlying the smart contract logic.

The figure 6.1 shows the proposed approach from extraction to transformation to XES logs.

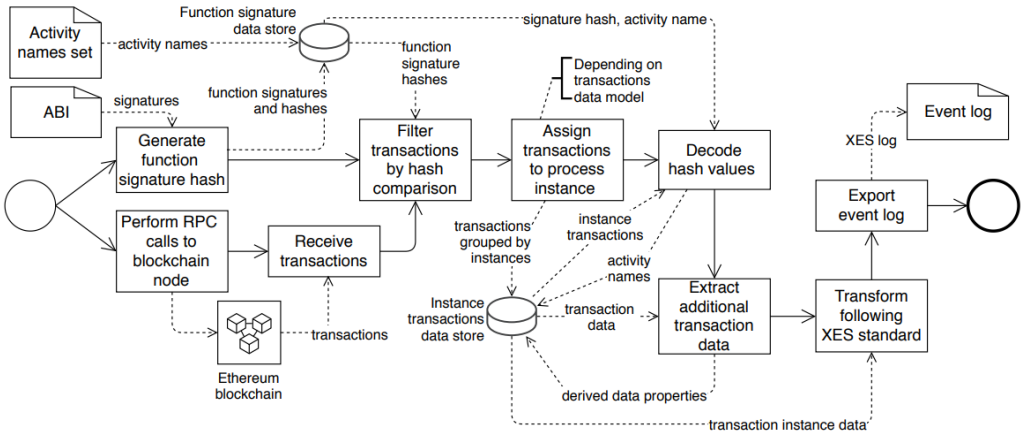

Figure 6.1: Overview of the methodology proposed in [15]

The proposed methodology starts with an activity dictionary containing the activity names, function signatures, and function selectors. An example of the dictionary is illustrated in figure 6.2. The function selector is obtained by computing the KECCAK[1] hash (i.e., the cryptographic hash function used by Ethereum) of the function signatures and saving the hexadecimal of its first 4 bytes. For instance, the task *"Customer has a problem"* is implemented by a function with signature *"Customer_Has_a_Problem()"*. The selector for such function is *0xefe73dcb*, because the KECCAK hash for *"Customer_Has_a_Problem()"* is ***0xefe73dcb****348c11a7ab31ce1620102e63c94e84ab393a78f187d1485c8a2c72cc*. The transaction denoting an activity enactment is selected using the function selector. Traces are built using the *"to"* field of blockchain transactions. Hence, every process instance refers to the transactions with a specific smart contract as the receiver.

The authors successfully evaluated the methodology on an Incident-management process [20] generating an XES log and applying to it the Inductive Visual Miner. It is evident that the methodology has huge limits in terms of flexibility and applies only to the intended target (i.e., blockchain-based BPMS). However, this limitation is intended by the authors who decided to focus specifically on blockchain-based BPMS.

[1] `https://keccak.team/index.html`

Activity name	Function signature	Function selector
Customer has a problem	`Customer_Has_a_Problem()`	`0xefe73dcb`
Get problem description	`Get_problem_description(int32 x)`	`0x92ed10ef`
Ask 1st level support	`Ask_1st_level_support(int32 y)`	`0x82b06df7`
Explain solution	`Explain_solution()`	`0x95c07f19`
Ask 2nd level support	`Ask_2nd_level_support()`	`0x63ad6b81`
Provide feedback for account manager	`Provide_feedback_for_account_manager()`	`0x58a66413`
Ask developer	`Ask_developer()`	`0xecb07b8c`
Provide feedback for 1st level support	`Provide_feedback_for_1st_level_support()`	`0x3b26a0ea`
Provide feedback for 2nd level support	`Provide_feedback_for_2nd_level_support()`	`0x9ec3200a`

Figure 6.2: Activity dictionary

6.2 Ethereum Logging Framework

The other analyzed papers are *"Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications"*[10] which describes the implementation of the Ethereum Logging Framework (ELF) and *"Process Mining on Blockchain Data: A Case Study of Augur"*[19] which describes in details a use case of the former. Weber et al. proposed a framework to extract process event data from DApps that utilize the Ethereum's transaction log as storage for logged data (i.e., Solidity events[2]).

The framework comprises three main parts:

- A *manifest* to specify the target smart contract addresses, the block range, and transformation rules (i.e., Ethereum logs → XES data)
- An *extractor* that applies the manifest rules
- A *generator* that produces Solidity code to emit events in line with the manifest specifications

To extract blockchain event data, it is necessary to provide a manifest containing the smart contract addresses, block range, and events to be analyzed. In addition, rules for transforming blockchain events into XES data can be specified to manipulate the data. This configuration step requires knowledge of the target smart contracts code and adds complexity to the data extraction process. As described in Chapter 2, blockchain events haven't standard parameters and are not mandatorily present in a function body. Nonetheless, the framework relies on the correct issuance of events. The Ethereum Logging Framework needs smart contracts to extensively emit events with enough data to create a correlation between them. Each event must contain a shared attribute key to build traces and must be emitted in the functions that we're interested in. To be precise, ELF does not require the shared keys to have the same name since it is possible to manipulate the data through the mapping. However, the meaning of the values identified by the keys should be the same to create a meaningful correlation. For instance, if a function lacks event emission, the framework will generate an XES log without that function and potentially lead to wrong results (e.g., process discovery model lacking an activity). This restricts the application of ELF to well-written smart contracts and compliant events. To partially address these issues, the authors provided a *generator* that produces Solidity code to emit events correctly.

[2] `https://docs.soliditylang.org/en/v0.8.16/contracts.html#events`

Nevertheless, this will work mainly for new smart contracts. It is unlikely that a team or an organization decides to deploy again an existing smart contract to make it compliant with the Ethereum Logging Framework. The deployment of a new smart contract on Ethereum costs thousands of dollars and requires all the services relying on the smart contract to be updated with the new address.

ELF was successfully evaluated on CryptoKitties[3] in [10] and Augur[4] in [19]. The evaluation on Augur is particularly interesting, as it identified a bug, albeit not a critical one, in the smart contract. This proves that the framework is valid but has limitations in the application to every smart contract.

6.2.1 Ethereum Logging Framework comparison

The Ethereum Logging Framework (ELF) was evaluated on CryptoKitties[5], a blockchain game developed on Ethereum that allows players to purchase, collect, breed, and sell virtual cats called cryptoKitties. Each cryptoKitty is represented by an identifier and its DNA, from which its features and appearance are derived. ELF requires the definition of a manifest where it should be specified the smart contract addresses, the block range, and a series of mappings to map Solidity events to XES events. Through the manifest, it is possible to select just a portion of events and map one of them to multiple XES events. Klinkmuller et al. selected the ID of a cryptoKitty as the process instance, so the lifecycle of a kitty is a trace.

In order to assess the proposed methodology in comparison with ELF, the former was executed on CryptoKitties using the same configurations:

- Smart contracts: SaleClockAuction([6](*0xb1690C08E213a35Ed9bAb7B318DE14420FB57d8C*) and KittyCore[7](*0x06012c8cf97BEaD5deAe237070F9587f8E7A266d*)
- Block range: *6605101 - 6618097*

The extraction process generated a log with 8936 events and 545 traces. The figures 6.3 and 6.4 show, respectively, the model discovered with the log generated with ELF and with our methodology. The first difference is the name and number of different activities present in the generated logs and discovered models.

- ELF: *AuctionCreated, AuctionSuccessful, AuctionCancelled, Birth, GiveBirht, Pregnant, KittybecameMother, Transfer, SireMadeSomeonePregnant, KittyBecameFather.*
- Proposed methodology: *bid, createGen0Auction, cancelAuction, createSaleAuction, bidOnSiringAuction, breedWithAuto, approve, transfer, createSiringAuction.*

As mentioned earlier, in ELF it is possible to specify mappings from one event to more than one. Klinkmuller et al. defined the followings:

- *Birth →Birth, GiveBirth, KittyBecameMother, KittyBecameFather*

[3] `https://www.cryptokitties.co/`
[4] `https://augur.net/`
[5] `https://www.cryptokitties.co/`
[6] `https://etherscan.io/address/0xb1690c08e213a35ed9bab7b318de14420fb57d8c`
[7] `https://etherscan.io/address/0x06012c8cf97bead5deae237070f9587f8e7a266d`

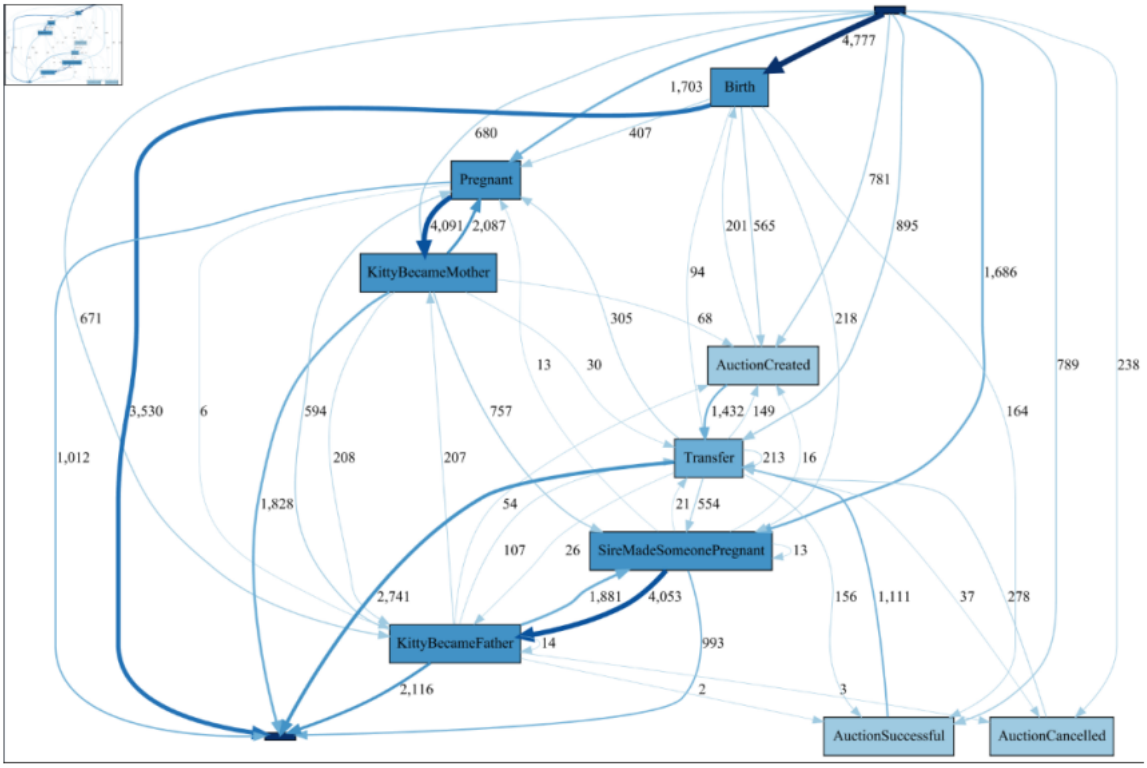

Figure 6.3: Heuristic Miner on CryptoKitties ELF-generated logs

- *Pregnant* →*Pregnant, SireMadeSomeonePregnant*

This justifies the difference in the number of activities between the two models. Another reason is the fact that more than one event can be emitted inside a Solidity function. Instead, regarding the name difference, it is simply explained by the fact that events and functions can have different names.

However, the mappings are worthy of more attention. The *Pregnant* event is emitted by the functions *breedWithAuto* and *bidOnSiringAuction*. The *Birth* event is emitted by the functions *giveBirth*, *createPromoKitty*, *createGen0Auction*. Even though some of the mentioned functions are not present in the generated logs due to the selected block range, it is clear that events are bound to functions. A function can be in the code even without an event, while events can be present only within a function body. Events can be emitted in multiple functions (e.g., *Birth* and *Pregnant*) and if not mapped to multiple XES events, there is a loss in terms of expressivity. An example is *Pregnant* which is mapped to *SireMadeSomeonePregnant* to comprehend the *bidOnSiringAuction* function. The proposed extraction methodology automatically catches two different events, while ELF needed the mapping on *Pregnant*. This exhibits that extraction logs from blockchain events require more configuration than transactions. Similarly, using the Ethereum Logging Framework is effort-prone and error-prone since it requires the definition of a manifest and knowledge of the smart contract's code under analysis. Nevertheless, mappings improve the readability of the events, e.g., *Birth* is surely clearer than *breedWithAuto*. Mappings could be an improvement that can be implemented in the proposed methodology since they do not require disruption in the methodology process. By introducing mappings, the methodology will also be able to catch function

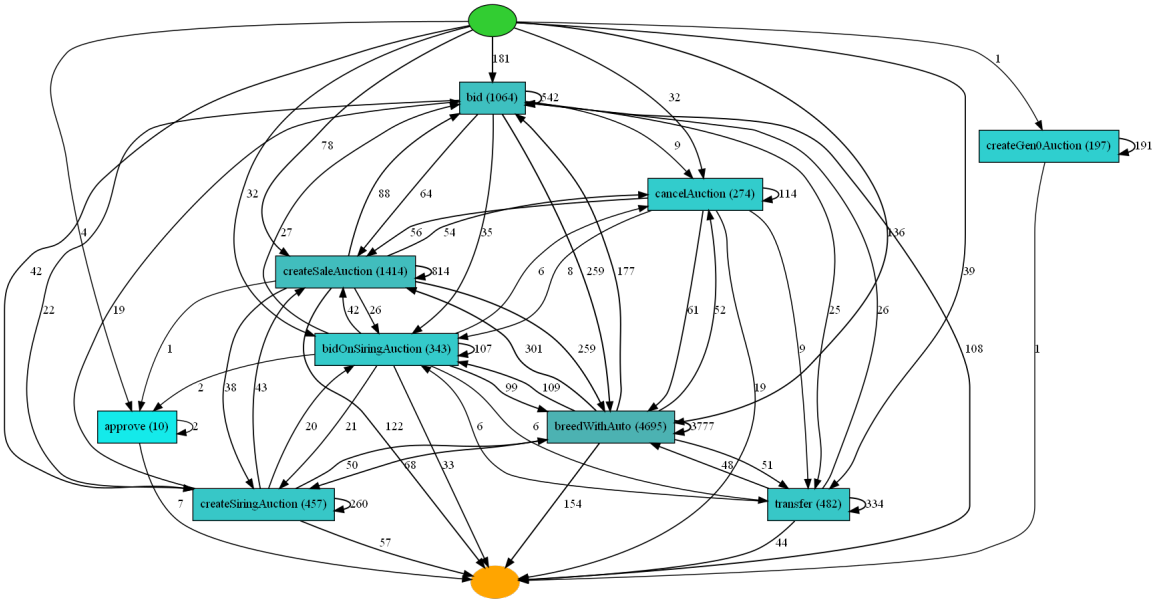

Figure 6.4: Heuristic Miner on CryptoKitties log generated with the proposed methodology

that emits more than one event. More on future works and possible improvements in Chapter 7.

7. Conclusions and future work

This last chapter concludes the research by describing the work done, the obtained results, and listing possible improvements and future work.

7.1 Conclusions

In this paper, a new extraction methodology was proposed to facilitate the task of extracting XES logs from blockchain data and applying process mining techniques to them. The main feature of the methodology is to be agnostic to the smart contract under analysis. The extraction focuses on blockchain transactions, which, unlike events, have standard parameters that do not change from smart contracts to smart contracts. The use case described on Decentraland demonstrates the applicability of the methodology in a real-world application. Process mining techniques such as process discovery and compliance checking were applied successfully to the generated logs. Of course, the list of applicable techniques is not limited to them: others, such as social network analysis and simulations, can also be applied to the logs. In addition to process mining, the statistics discovered in Chapter 5 demonstrate that XES logs can serve other purposes and that the proposed methodology is not limited to process mining. Analyzing the internal transactions with process mining is something that, to our knowledge, has never been done before. Internal transactions are difficult to retrieve because they are not stored on-chain. However, the ability to inspect the execution trace of immutable applications, such as smart contracts, can help discover bugs and prevent exploits, especially in the context of blockchain, where money (e.g., cryptocurrency, decentralized finance) and, in general, values are managed.

7.2 Future work

Even though we're satisfied with the obtained results, there is of course room for additional research.

- *Extend the research on Decentraland trying different combinations of fields as case ID, sorting, and concept name*: we took the viewpoint of a user (i.e., address) selecting the *from* field as case ID. Selecting different values to construct traces could lead to interesting results and insights that, in case, could be reported to Decentraland developers similarly to what has been done by Weber et al. in [19].

- *Validate the methodology for multiple use cases*: we ran the methodology extensively on Decentraland and CryptoKitties only for comparison with ELF. The methodology should be validated on more use cases from different sectors, such

as healthcare, manufacturing, the internet of things, decentralized finance, and blockchain-based games.

- *Continue the validation with existing process discovery algorithms*: we limited the application of process discovery algorithms to the Inductive Miner, Heuristic Miner, and Alpha algorithm. The generated XES logs should be tested and evaluated with other process discovery algorithms to find the most accurate for blockchain-based applications. This activity could comprehend the definition of a new process discovery algorithm able to deal natively with blockchain-based applications.

- *Extend the research on internal transactions*: we applied process discovery algorithms to the XES logs generated from internal transactions. It could be interesting to deepen the research on such data by analyzing the execution trace and the eventual variants.

- *Improve the usability and performances of the methodology*: we limited the research to demonstrate the applicability of the extraction methodology. From now on, we can focus on the implementation of additional features to improve the usability and performance of the methodology. For instance, the mapping of the fields in the log to one or more values similar to what has been done on ELF [10] can enhance the expressivity of the fields and permits complex manipulations.

Bibliography

[1] Wil Aalst, A. Weijters, and Laura Măruşter. "Workflow Mining: Discovering Process Models from Event Logs". In: *Knowledge and Data Engineering, IEEE Transactions on* 16 (Oct. 2004), pp. 1128 –1142. DOI: 10.1109/TKDE.2004.47.

[2] Paul Beck et al. "BLF: A Blockchain Logging Framework for Mining Blockchain Data". In: (Sept. 2021).

[3] Alessandro Berti and Wil M.P. van der Aalst. "Reviving Token-based Replay: Increasing Speed While Improving Diagnostics". In: *ATAED@Petri Nets/ACSD*. 2019.

[4] Fabian Rojas Blum. "Metrics in process discovery". In: 2015.

[5] J. Buijs, B. Dongen, and Wil Aalst. "Quality Dimensions in Process Discovery: The Importance of Fitness, Precision, Generalization and Simplicity". In: *International Journal of Cooperative Information Systems* 23 (Mar. 2014), p. 1440001. DOI: 10.1142/S0218843014400012.

[6] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf. [Online; accessed 16-September-2022]. 2014.

[7] Raffaele Conforti et al. "Beyond Tasks and Gateways: Discovering BPMN Models with Subprocesses, Boundary Events and Activity Markers". In: *Business Process Management*. Ed. by Shazia Sadiq, Pnina Soffer, and Hagen Völzer. Cham: Springer International Publishing, 2014, pp. 101–117. ISBN: 978-3-319-10172-9.

[8] Frank Duchmann and Agnes Koschmider. "Validation of Smart Contracts Using Process Mining". In: (Jan. 2019).

[9] IEEE. "IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams". In: *IEEE Std 1849-2016* (Nov 2016).

[10] Christopher Klinkmüller et al. "Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications". In: *Business Process Management: Blockchain and Central and Eastern Europe Forum*. Ed. by Claudio Di Ciccio et al. Cham: Springer International Publishing, 2019, pp. 71–86.

[11] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. "Discovering Block-Structured Process Models from Event Logs - A Constructive Approach". In: *Application and Theory of Petri Nets and Concurrency*. Ed. by José-Manuel Colom and Jörg Desel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 311–329. ISBN: 978-3-642-38697-8.

[12] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. "Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour". In: *Business Process Management Workshops*. Ed. by Niels Lohmann, Minseok Song, and Petia Wohed. Cham: Springer International Publishing, 2014, pp. 66–78. ISBN: 978-3-319-06257-0.

[13] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. "Scalable Process Discovery with Guarantees". In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Khaled Gaaloul et al. Cham: Springer International Publishing, 2015, pp. 85–101. ISBN: 978-3-319-19237-6.

[14] Orlenys López-Pintado et al. "Caterpillar: A business process execution engine on the Ethereum blockchain". In: *Software: Practice and Experience* 49 (2019), pp. 1162 –1193.

[15] Roman Mühlberger et al. "Extracting Event Logs for Process Mining from Data Stored on the Blockchain". In: *Business Process Management Workshops*. Ed. by Chiara Di Francescomarino, Remco Dijkman, and Uwe Zdun. Cham: Springer International Publishing, 2019, pp. 690–703. ISBN: 978-3-030-37453-2.

[16] Jorge Muñoz-Gama and Josep Carmona. "A Fresh Look at Precision in Process Conformance". In: *Business Process Management*. Ed. by Richard Hull, Jan Mendling, and Stefan Tai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 211–226. ISBN: 978-3-642-15618-2.

[17] Esteban Ordano et al. *Decentraland: a blockchain-based virtual world*. `https://decentraland.org/whitepaper.pdf`. [Online; accessed 05-September-2022]. 2021.

[18] An Binh Tran, Qinghua Lu, and Ingo Weber. "Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management". In: *BPM*. 2018.

[19] Ingo Weber et al. "Process Mining on Blockchain Data: A Case Study of Augur". In: *Business Process Management*. Ed. by Artem Polyvyanyy et al. Cham: Springer International Publishing, 2021, pp. 306–323.

[20] Ingo Weber et al. "Untrusted Business Process Monitoring and Execution Using Blockchain". In: *Business Process Management*. Ed. by Marcello La Rosa, Peter Loos, and Oscar Pastor. Cham: Springer International Publishing, 2016, pp. 329–347.

[21] A. J. M. M. Weijters, Wil M.P. van der Aalst, and Ana K. A. de Medeiros. "Process mining with the HeuristicsMiner algorithm". In: 2006.

[22] J. M. E. M. van der Werf et al. "Process Discovery Using Integer Linear Programming". In: *Applications and Theory of Petri Nets*. Ed. by Kees M. van Hee and Rüdiger Valk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 368–387. ISBN: 978-3-540-68746-7.

Acknowledgments

First and foremost, I would like to thank my thesis supervisors, Barbara Re and Andrea Morichetta, for their time and guidance over the past few months. This research would not have been possible without their knowledge and experience at every step.

I would also like to thank the University of Camerino and the Computer Science faculty for giving me the opportunity to approach the world of research through this experimental thesis.