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

Through 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 strategies cannot. However, due to the structure of blockchain data, applying Process Mining in such a context is challenging. The techniques created so far by researchers have limitations when applied to smart contracts. Some are tailored to specific use cases like blockchain-based Business Process Management Systems. Others require the data under analysis to be in a precise format which is uncommon for a significant portion of the existing smart contracts. To solve this challenge, we propose an application-agnostic extraction methodology 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 execution. 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 comprises five steps: (i) extraction of data from smart contracts, (ii) cleaning of raw data, (iii) selecting and defining sorting criteria, (iv) trace construction, and, finally, (v) XES log generation. 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 from different block ranges and one or more Decentraland smart contracts combined. Unlike event data collection, a plain execution with transactions does not require prior knowledge of smart contract code. On the generated XES logs, we successfully applied several Process Mining techniques like Simulation, Social Network Analysis, and Process discovery.

Contents

Listings

List of Figures

List of Tables

1. Introduction

In recent years, blockchain has been acquiring attention as a 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. Process Mining techniques have been vastly employed on traditional systems to exploit and deeply understand business processes. Examples are: compare a process's actual execution with the intended one (i.e., conformance checking); produce high-level models from process logs (i.e., process discovery); analyze business parties' interactions (i.e., social network analysis). In theory, it should be possible to achieve the same results on business processes executed on the blockchain. However, due to the structure of blockchain data, applying Process Mining in 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 timestamp. In recent years, researchers have proposed multiple techniques to ease this task. Yet, there are limitations in the application to smart contracts structurally different from the intended target. For instance, [14] is tailored to blockchain-based Business Process Management Systems. This kind of system is generated from a BPMN model. Hence, we know a priori the context, the activities, and the execution flow of the process under analysis. These assumptions do not hold for every smart contract in the blockchain. Other techniques, like [8] and [6], focus on events. Events are logs issued during the execution of smart contract functions. Events don't have standard parameters and are not mandatorily present in a function body. To enable Process Mining, they must contain a shared parameter (i.e., case ID) and be emitted correctly and in the right place.

To solve this challenge, we propose an application-agnostic extraction methodology to collect data from every smart contract and enable the application of Process Mining techniques. The proposed methodology focuses on blockchain transactions and their internal execution. The transactions, unlike events, have standard parameters that ensure a common ground of operations. In every transaction, we can rely on the fact that the *hash*, the *sender*, the *receiver*, and other parameters are present. The transactions' internal execution represents 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 user selects the smart contracts and, optionally, the block range for the data retrieval.

- *Raw data cleaning*: the transaction parameters are converted into a readable format since some are hexadecimal. Moreover, unnecessary parameters are removed.
- *Select and define sorting criteria*: as mentioned above, timestamps of transactions in the same block are equal. We need additional or different criteria to sort transactions.
- *Traces construction*: a parameter is selected as case ID to construct traces.
- *XES log generation*

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 chose Decentraland because Process Mining has not been applied extensively to metaverses and the variety of functions it offers. (e.g., cryptocurrencies and NFTs exchange, virtual lands, games, and more). We were able to generate XES logs from different block ranges and one or more Decentraland smart contracts combined. On the generated XES logs, we successfully applied and evaluated several Process Mining techniques like Process Discovery, Simulation, and Social Network Analysis. Special attention was given to the logs generated from the transactions' internal execution on which we applied BPMN Miner [5], a novel technique able to output a BPMN model with subprocesses inside.

In the following, we first introduce relevant background information on the blockchain, Ethereum, process mining, and XES logs in Section 2. The related work is described in Section 3. The proposed methodology is introduced in Section 4, implemented in Section 5, and evaluated in Section 6. Finally, Section 7 concludes and lists future challenges.

2. Background

2.1 Blockchain

A blockchain is a growing list of records, called blocks, that are securely linked together using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. As each block contains information about the block previous to it, they form a chain, with each additional block reinforcing the ones before it. Therefore, blockchains are resistant to modification of their data because once recorded, the data in any given block cannot be altered retroactively without altering all subsequent blocks. Blockchains are typically managed by a peer-to-peer network where nodes collectively adhere to a protocol to communicate and validate new blocks.

2.1.1 Ethereum

Ethereum is a decentralized, open-source blockchain with smart contract functionality. 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 simply 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 the next action when specific conditions are met. 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 (i.e., 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, 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 – 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 transaction, binary code to create 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 Transactions internal execution

A transaction's internal execution, also called internal transaction, is the consequence of smart contract logic triggered by an external transaction. A smart contract engagement can result in tens or hundreds of internal transactions. These represent value transfers that occur 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. Transaction's internal executions 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 transactions internal execution using EthTx[1] which is an advanced decoder for blockchain transaction 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": "0
        x15ee97483fdada5e3ec49991f4de338e554c18b44a59fb3612d84faf99b52dcd
        ",
    "CALL_ID": "\\N",
```

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

```
    "CALL_TYPE": "call",
    "FROM_ADDRESS": "0x28cd504f564c7288413148a78788e29e7fffebf0
        ",
    "FROM_NAME": "0x28cd504f564c7288413148a78788e29e7fffebf0",
    "TO_ADDRESS": "0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d",
    "TO_NAME": "LAND",
    "FUNCTION_SIGNATURE": 22465,
    "FUNCTION_NAME": "setApprovalForAll",
    "VALUE": 0,
    "ARGUMENTS": "{\"address\":\"0
        x8e5660b4ab70168b5a6feea0e0315cb49c8cd539\",\"approved
        \":\"True\"}",
    "RAW_ARGUMENTS": "[{\"name\":\"address\",\"raw\":\"0
        x0000000000000000000000008e5660b4ab70168b5a6feea0e0315cb49c8cd539
        \",\"type\":\"address\"},{\"name\":\"approved\",\"raw
        \":\"0
        x0000000000000000000000000000000000000000000000000000000000000001
        \",\"type\":\"bool\"}]",
    "OUTPUTS": "{}",
    "RAW_OUTPUTS": "[]",
    "GAS_USED": 29105,
    "ERROR": null,
    "STATUS": true,
    "ORDER_INDEX": 1315,
    "DECODING_STATUS": true,
    "STORAGE_ADDRESS": "0
        xf87e31492faf9a91b02ee0deaad50d51d56d5d4d"
}
```

The *CALL_ID* field allows for distinguishing a transaction from the internal transaction execution. When it is \\N it indicates that the record is a transaction. Instead, for subcalls, it can assume different values indicating deeper levels in the trace. Example are: *0*, *0_0* (subcall 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, we can get the full execution trace of a 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 code example that shows the use of events to log that the value has changed.

```
pragma solidity 0.5.17;
```

```
contract Counter {

    event ValueChanged(uint oldValue, uint256 newValue);

    // Private variable of type unsigned int to keep the
       number of counts
    uint256 private count = 0;

    // Function that increments our counter
    function increment() public {
        count += 1;
        emit ValueChanged(count - 1, count);
    }

    // Getter to get the count value
    function getCount() public view returns (uint256) {
        return count;
    }

}
```

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

2.2.2 Any other techniques

Add a subsection for each applied technique.

2.3 XES logs

The XES [7] 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 log and extensions is part of the standard. A 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 for Process Mining.

A XES file is structured as follows. On the top level there is one **log** object which contains all event information related to one specific process. A log contains an arbitrary number of **trace** objects. Each trace describes the execution of one specific instance, or case, of the logged process. Every trace contains an arbitrary number of event **objects**. 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.

```
<log xes.version="1.0" xes.features="nested-attributes"
    openxes.version="1.0RC7" xmlns="http://www.xes-standard.org
    /">
        <string key="concept:name" value="TEST"/>
        <string key="lifecycle:model" value="standard"/>
        <trace>
                <string key="concept:name" value="1411"/>
                <event>
                        <string key="id" value="1411"/>
                        <string key="org:resource" value="user
                            "/>
                        <date key="time:timestamp" value
                            ="1970-03-27T12:02:00+10:00"/>
                        <string key="lifecycle:transition"
                            value="complete"/>
                        <string key="concept:name" value="A"/>
                </event>
                <event>
                        <string key="id" value="1411"/>
                        <string key="org:resource" value="user
                            "/>
                        <date key="time:timestamp" value
                            ="1970-03-27T12:32:00+10:00"/>
                        <string key="lifecycle:transition"
                            value="complete"/>
                        <string key="concept:name" value="S1A
                            "/>
                        <string key="ids1" value="2S2505"/>
                </event>
        </trace>
</log>
```

3. Related work

Before starting my research on Process Mining and blockchain, I decided to examine the existing work on the topic. I searched on Google Scholar using the text parameter *"process mining blockchain"*, and I found out that in recent years, researchers have proposed multiple methodologies to apply Process Mining to blockchain data. To skim the obtained results, I further analyzed only the papers that respected the following criteria:

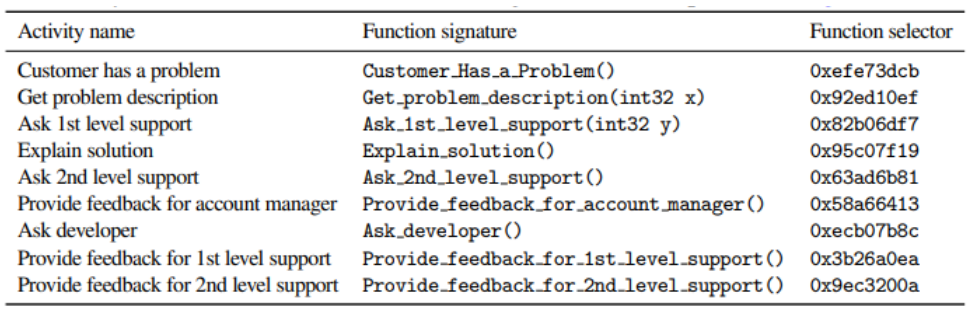
- Implement an extraction methodology
- Deal with the Ethereum blockchain
- Considerable number of citations (i.e., *>100*)

After the filtering, the remaining papers were: *Extracting Event Logs for Process Mining from Data Stored on the Blockchain*[14], *Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications*[8], and *Process Mining on Blockchain Data: A Case Study of Augur*[6]. The examination of the mentioned works highlighted limitations and room for additional research. The following will describe the three papers and their limitations.

3.1 Process Mining on Blockchain-based BPMS

Di Ciccio et al. in [14] defined a methodology to extract logs from the transactions involving smart contracts generated with a blockchain-based Business Process Management System (BPMS) like Caterpillar[12] and Lorikeet[16]. This kind of system generates smart contracts code from BPMN or Choreography models. Hence we know 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, we would not have access to the models underlying the smart contract logic. The proposed methodology starts from an activity dictionary containing the activity names, function signatures, and other information. An example of the dictionary is shown in figure 3.1. Traces are built using the *transaction.to* field. 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 [17]. 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.



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 3.1: Activity dictionary

3.2 Ethereum Logging Framework (ELF)

The other analyzed papers are *Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications*[8] and the respective use case[6]. 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[1]). The framework includes (i) a *manifest* to specify transformation rules (i.e., Ethereum logs → XES data), (ii) an extractor that applies the manifest rules, and (iii) a *generator* that produces Solidity code to emit events in line with the manifest specifications. To extract data, the user must create a manifest containing the smart contract addresses, the block range, and the events he wants to analyze. Additionally, he needs to specify mappings from the events data to XES data. This configuration step requires knowledge of the target smart contracts code and adds complexity to the data extraction process. As described in Section 2, events haven't standard parameters and are not mandatorily present in a function body. Nonetheless, the framework relies on the correct issuance of events. ELF 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. 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. However, this will work mainly for new smart contracts. It is unlikely that a team 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[2] and Augur[3][6]. 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.

The table 3.1 presents an overview of the analyzed papers with the *methodology*, the *target data*, the *target application*, *case ID*, and the *process discovery algorithm* since

[1]https://docs.soliditylang.org/en/v0.8.16/contracts.html#events
[2]https://www.cryptokitties.co/
[3]https://augur.net/

it was applied in all the three papers.

Methodology	**Data**	**Target**	**Case ID**	**Discovery algorithm**
ELF[8]	Event	CryptoKitties	Kitty ID	DFGs
ELF[6]	Event	Augur	Market ID	Inductive Miner
BPMS[14]	Transaction	Incident-management[17]	transaction.to	Inductive Visual Miner

Table 3.1: Related work overview

4. Methodology

The analysis of the related work in Section 3 shows that the present research is missing an application-agnostic methodology to enable the application of Process Mining techniques on blockchains easily. A methodology capable of collecting data from every smart contract and generating XES logs ready for 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 Section 2, have standard parameters and, for this reason, have been selected as the input data of the methodology. In addition to transactions, the methodology will focus on transactions' internal execution to exploit the execution trace of smart contract functions through Process Mining. Transaction internal execution has been as well described in Section 2.

The methodology has been implemented to work with three different inputs:

- transactions: collect transactions from one or more smart contracts and generate a XES log according to the parameter select by the user: *sort by*, *case ID*, and *concept name.*
- transactions internal execution: collect transactions internal execution from one or more smart contracts and generate an XES log for each different top-level function name. Each log refers to a specific function, and its traces, which contain internal executions, are grouped by transaction hash.
- combination of the two: by combining transactions and the respective internal execution we can obtain logs that contain sub-processes. To exploit such logs, we can use BPMN Miner [5], a novel technique able to output a BPMN model with sub-processes inside.

The proposed methodology, as depicted in the figure 4.1, consists of five steps (i.e., *data extraction*, *raw data cleaning*, *sorting*, *trace construction*, and *XES generation*) that are described in the following sections.

4.1 Data extraction

The *data extraction* step is the first step of the methodology. As the name implies, blockchain data is extracted according to two values provided by the user: smart contract addresses and block range. The former indicates the addresses we want to collect data about. The latter is optional and is used to scope the research in a precise time frame. More specifically, the methodology requires two addresses for each smart contract: a transaction address from which it will collect transactions, and an ABI address from which will be used to get the ABI. More on that in the following sections. The

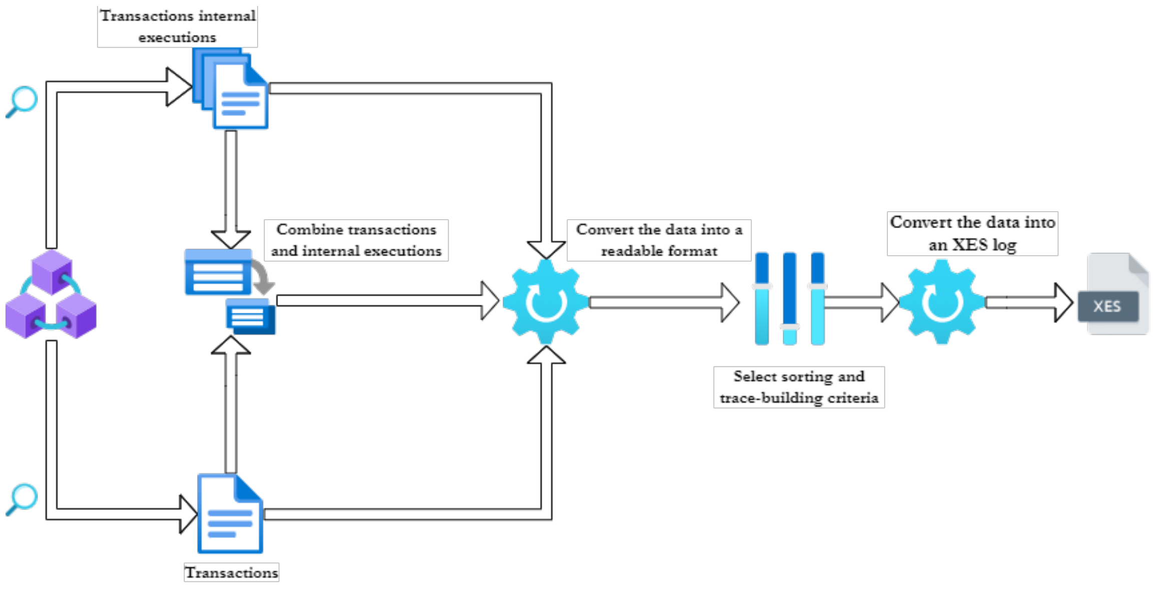

Transactions internal executions
Combine transactions and internal executions
Convert the data into a readable format
Convert the data into an XES log
XES
Select sorting and trace-building criteria
Transactions

Figure 4.1: Overview of the methodology

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

4.1.1 Transactions

The collected transactions are in the format described in Section 2.1.2. Transactions can be collected using blockchain explorers like Etherscan [1].

Algorithm 1 Transactions retrieval

```
Require: $contractsAddresses, startBlock, endBlock$
1: for all $address$ in $contractsAddresses$ do
2:     $transactions = etherscan.getTxsByAddress(address, startBlock, endBlock)$
3: end for
```

4.1.2 Transactions internal execution

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[2], 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 transactions' internal execution is similar to Algorithm 1 with the only difference of EthTx instead of Etherscan. The data obtained is in the format shown in Section 2.1.3.

[1]https://etherscan.io/
[2]https://ethtx.info/

4.2 Raw data cleaning

The collected data is in a raw format. Some fields are encoded in hexadecimal according to the smart contract Application Binary Interface (ABI) [3]. To make such fields readable, we need the ABI of the smart contract. The ABI is the interface of the smart contract. It is the de facto method for encoding Solidity contract calls for the EVM and backward. Earlier we mentioned that the methodology requires two addresses: a transaction address, and an ABI address. In most cases, these two addresses coincide, but a smart contract may be developed with the proxy pattern[4] 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[5] and use the ABI of the main contract[6] to decode the hexadecimal fields.

One of the encoded fields is *transaction.input* which contains the executed function name and the respective parameters.

Algorithm 2 Data decoding

```
1: $contractABI = getContractABI(contract)$
2: for all $transaction$ in $transactions$ do
3:     $function, functionParams = decodeInput(contractABI, transaction.input)$
4:     $transaction.function = function$
5:     $transaction.functionParams = functionParams$
6: end for
```

In addition to the operation described above, fields not needed in the next steps are removed.

The decoding step is not performed on the internal transaction since the data obtained with EthTx contains both the raw data and the decoded data. Instead, unnecessary fields are removed also there.

4.3 Sorting

As mentioned in Section 1, timestamps are not accurate because transactions mined in the same block have the same timestamp. In blockchains, transactions are executed when they are validated and put inside a block, not when the users 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. The methodology, by default, sorts transactions by *timestamp* and *transactionIndex*. However, it also supports sorting by other columns selected by the user.

The sorting step is shared for transactions and transactions' internal execution.

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

4.4 Trace construction

The trace construction step is a crucial step for the output of Process Mining techniques executed on the generated XES log. The selection of the case ID strongly influence the outcome of the techniques and allows to highlights different aspects.

As mentioned earlier, the methodology can work with 3 different input data: transactions, transactions internal execution, and a combination of the two. These three inputs differs in the trace construction step.

4.4.1 Transactions

For the trace construction step on transactions, we just need to select a field as the case ID. This field will influence the outcome of the Process Mining techniques that will be applied to the generated log. For instance, if we select the field *from* (i.e., transaction sender), we will put in a trace all the transactions sent by a user, and by running Process Discovery we will get a model representing the lifecycle of transactions sent by the user towards a specific contract.

4.4.2 Transactions internal execution

For transaction internal execution the situation is a little trickier. As mentioned above, we want 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, we first need to group internal transactions 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 we can use the *CALL_ID* field. Records with *CALL_ID == "\\N"* are the parent transactions. We can iterate through the records and store the function name in correspondence of *CALLID == "\\N"*. In the same iteration, we mark the internal transactions with the function name in order to identify and group them. We also prefix the function name of internal transactions with the name of the smart contract executing: {*contract_name*}_{*function_name*}. It is done to identify the smart contract that performs a function, as it may happen that function names are similar among smart contracts. Now we're set to split the records into different sets where each contains the internal transactions for a specific function name.

4.4.3 Combination of transactions and internal execution

For the combination of transaction and internal execution, the procedure is again different. As said earlier, the end goal here is to execute the BPMN Miner algorithm, and to make the dataset compliant with it we need to make some modifications to the records. 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 subprocess to parent process instances (i.e., foreign keys). For an in-depth explanation of BPMN Miner refer to [5].

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 (e.g., *"function_name": "0x....123"*). In this way, we add a relation between internal transactions that allows 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*), and the concept name of records with *CALL_ID != "\\N"* is prefixed with {*function_name*}_{*from*} (e.g., *approve_LAND.approve*) 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 subcalls to group them in traces. This column contains the address of the user invoking the "top-level" transaction. In the end, we have two groupings in the data: (i) the id field used to group records in traces, and the new field to identify subprocesses. Moreover, since BPMN Miner allows us to select the keys for which we want to find subprocesses, 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. This modifications have been done using as reference the artificial log contained in the BPMN Miner source code[7] and also mentioned in [5].

4.5 XES generation

After the trace construction step we are finally ready to generate the XES log.

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

5. Framework

5.1 Framework

The methodology described in Section 4 has been implemented with Python and integrated into a client-server application for end users' usage. The application allows to collect transactions and generate XES logs from one or more smart contracts and a block range. It is availaible at `https://xes-ethereum-extractor.herokuapp.com/`. The framework source code is publicly available in the project's GitHub repository[1]. The following sections describe the tools employed in the framework development.

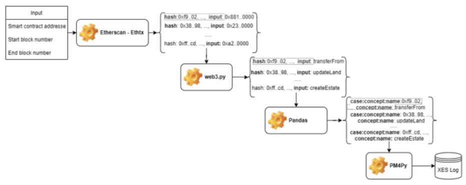

Input
Smart contract addresse
Start block number
End block number
Etherscan - Ethtx
hash:0xf9..02, ..., input: 0x881..0000
hash: 0x38..98, ..., input: 0x23..0000
hash: 0xff..cd, ..., input: 0xa2..0000
web3.py
hash:0xf9..02, ..., input: transferFrom
hash: 0x38..98, ..., input: updateLand
hash: 0xff..cd, ..., input: createEstate
Pandas
case:concept:name:0xf9..02, ..., concept:name: transferFrom
case:concept:name: 0x38..98, ..., concept:name: updateLand
case:concept:name: 0xff..cd, ..., concept:name: createEstate
PM4Py
XES Log

Figure 5.1: Framework

5.1.1 Etherscan

Etherscan[2] is the leading blockchain 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[3] 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 used exclusively for transaction data, not for transactions' internal execution.

5.1.2 Ethtx

EthTx Transaction Decoder[4] 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. EthTx is also available as a Python library and has been used to collect transactions' internal execution data.

[1]`https://github.com/yuripaoloni/xes-ethereum-extractor`
[2]`https://etherscan.io/`
[3]`https://docs.etherscan.io/misc-tools-and-utilities/using-this-docs`
[4]`https://tokenflow.live/products/ethtx`

5.1.3 Web3.py

Web3.py[5] is a 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.

5.1.4 Pandas

Pandas[6] 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 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.

5.1.5 PM4Py

PM4Py[7] 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 as described later in Section 6.

[5] `https://web3py.readthedocs.io/en/stable/`
[6] `https://pandas.pydata.org/`
[7] `https://pm4py.fit.fraunhofer.de/`

5.1.6 ProM

ProM[8] 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[5]. ProM has been utilized to test the XES log generated with the combination of transactions and their internal executions. The results are described in Section 6.

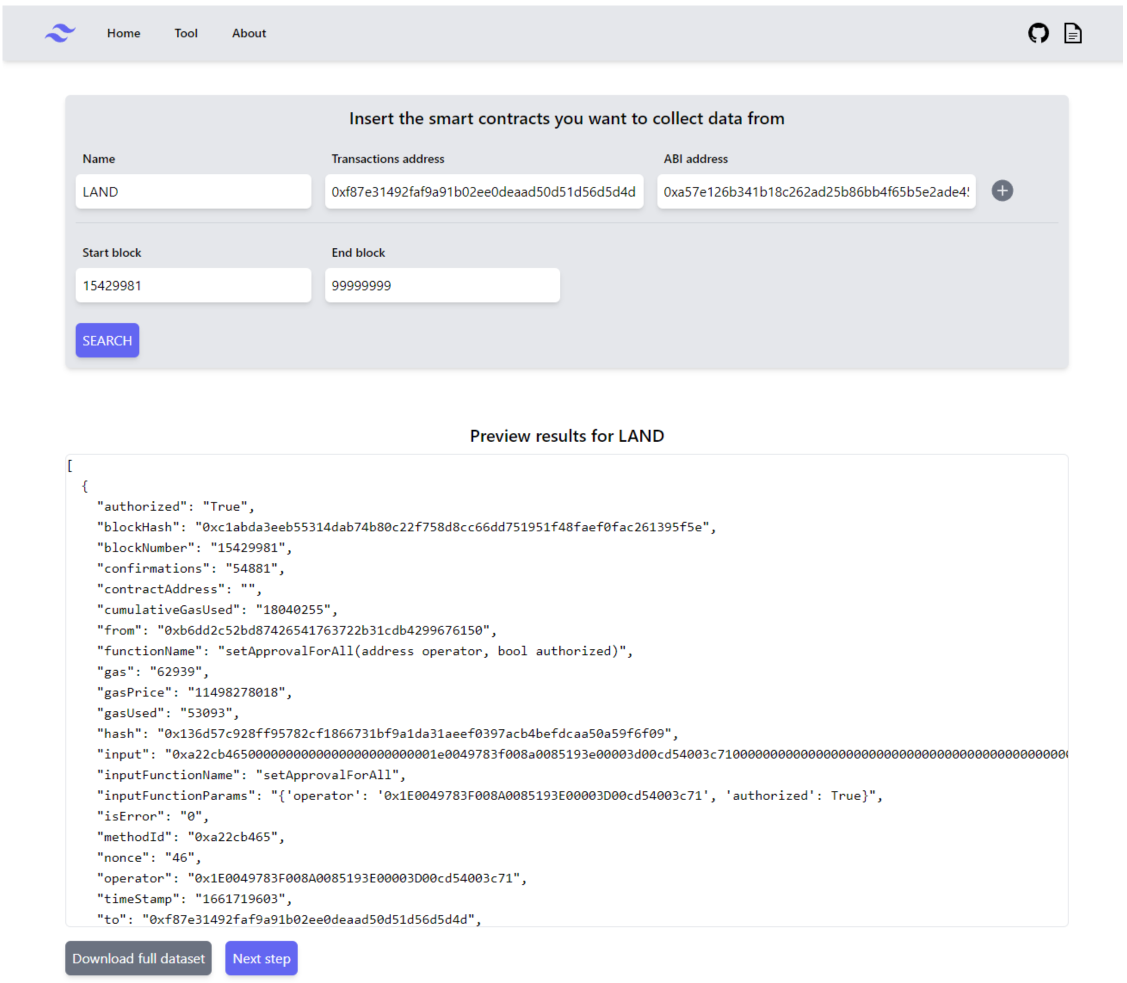
5.1.7 Flask

Flask[9] 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 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.

5.1.8 Next.js

Next.js[10] is a React framework that enables several extra features, including server-side rendering and generating static websites. React[11] 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.

[8]`https://www.promtools.org/doku.php`
[9]`https://flask.palletsprojects.com/en/2.2.x/`
[10]`https://nextjs.org/`
[11]`https://reactjs.org/`



Insert the smart contracts you want to collect data from

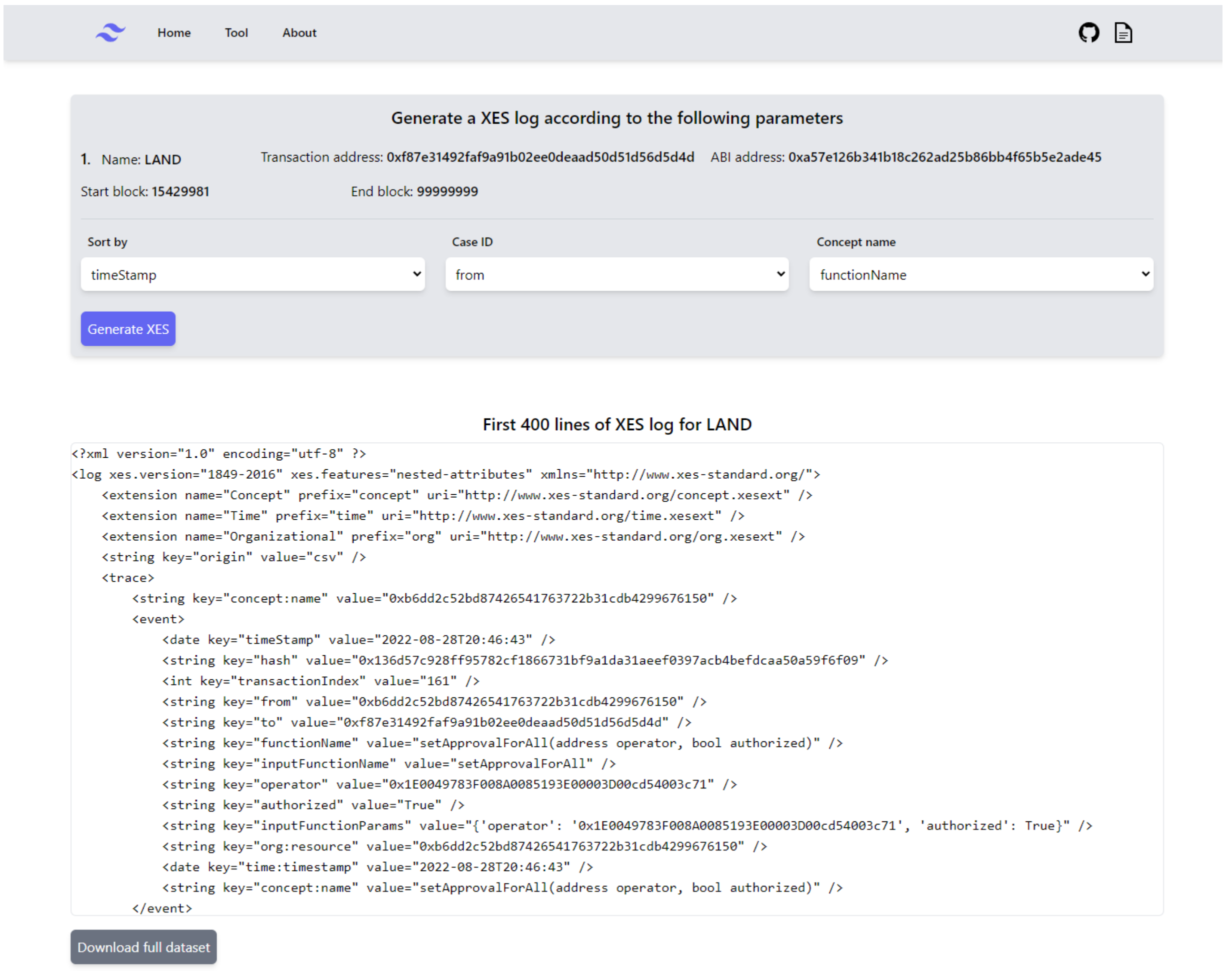
Name	Transactions address	ABI address
LAND	0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d	0xa57e126b341b18c262ad25b86bb4f65b5e2ade4!

Start block	End block
15429981	99999999

Preview results for LAND

```
[
  {
    "authorized": "True",
    "blockHash": "0xc1abda3eeb55314dab74b80c22f758d8cc66dd751951f48faef0fac261395f5e",
    "blockNumber": "15429981",
    "confirmations": "54881",
    "contractAddress": "",
    "cumulativeGasUsed": "18040255",
    "from": "0xb6dd2c52bd87426541763722b31cdb4299676150",
    "functionName": "setApprovalForAll(address operator, bool authorized)",
    "gas": "62939",
    "gasPrice": "11498278018",
    "gasUsed": "53093",
    "hash": "0x136d57c928ff95782cf1866731bf9a1da31aeef0397acb4befdcaa50a59f6f09",
    "input": "0xa22cb4650000000000000000000000001e0049783f008a0085193e00003d00cd54003c7100000000000000000000000000000000000000000
    "inputFunctionName": "setApprovalForAll",
    "inputFunctionParams": "{'operator': '0x1E0049783F008A0085193E00003D00cd54003c71', 'authorized': True}",
    "isError": "0",
    "methodId": "0xa22cb465",
    "nonce": "46",
    "operator": "0x1E0049783F008A0085193E00003D00cd54003c71",
    "timeStamp": "1661719603",
    "to": "0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d",
```

Figure 5.2: Transactions collection page



Generate a XES log according to the following parameters

1. Name: **LAND** Transaction address: **0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d** ABI address: **0xa57e126b341b18c262ad25b86bb4f65b5e2ade45**

Start block: **15429981** End block: **99999999**

Sort by: timeStamp

Case ID: from

Concept name: functionName

First 400 lines of XES log for LAND

```
<?xml version="1.0" encoding="utf-8" ?>
<log xes.version="1849-2016" xes.features="nested-attributes" xmlns="http://www.xes-standard.org/">
    <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext" />
    <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext" />
    <extension name="Organizational" prefix="org" uri="http://www.xes-standard.org/org.xesext" />
    <string key="origin" value="csv" />
    <trace>
        <string key="concept:name" value="0xb6dd2c52bd87426541763722b31cdb4299676150" />
        <event>
            <date key="timeStamp" value="2022-08-28T20:46:43" />
            <string key="hash" value="0x136d57c928ff95782cf1866731bf9a1da31aeef0397acb4befdcaa50a59f6f09" />
            <int key="transactionIndex" value="161" />
            <string key="from" value="0xb6dd2c52bd87426541763722b31cdb4299676150" />
            <string key="to" value="0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d" />
            <string key="functionName" value="setApprovalForAll(address operator, bool authorized)" />
            <string key="inputFunctionName" value="setApprovalForAll" />
            <string key="operator" value="0x1E0049783F008A0085193E00003D00cd54003c71" />
            <string key="authorized" value="True" />
            <string key="inputFunctionParams" value="{'operator': '0x1E0049783F008A0085193E00003D00cd54003c71', 'authorized': True}" />
            <string key="org:resource" value="0xb6dd2c52bd87426541763722b31cdb4299676150" />
            <date key="time:timestamp" value="2022-08-28T20:46:43" />
            <string key="concept:name" value="setApprovalForAll(address operator, bool authorized)" />
        </event>
```

Figure 5.3: XES generation page

6. Use case

To demonstrate the validity of the proposed methodology, we decided to execute it on Decentraland, a metaverse and digital assets marketplace developed on the Ethereum blockchain. We were able to generate XES logs from different block ranges and one or more smart contracts combined. Moreover, we successfully executed Process Mining techniques on them and evaluated the results with Conformance Checking. The following sections describe in detail the methodology application.

6.1 Decentraland

Decentraland [15] 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 that 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.

Speaking more technically, Decentraland has 4 main contracts:

- MANA[1]. It is the ERC20[2] smart contract that manages the MANA token. MANA is Decentraland's cryptocurrency which is used to buy LANDs, ESTATEs, and any other collectibles of the ecosystem. It is also used to vote 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 and metadata (version, name, description, ipns). It provides functions to transfers parcels (i.e., *transferFrom*, *transferLand*), create ESTATEs (i.e., *createEstate*), updated parcels metadata (i.e., *updateLandData*), and others.

[1] https://etherscan.io/token/0x0f5d2fb29fb7d3cfee444a200298f468908cc942

[2] https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

[3] https://etherscan.io/address/0xa57e126b341b18c262ad25b86bb4f65b5e2ade45

- 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.

Figure 6.1: Decentraland logo

We decided to select Decentraland as the use case for three reasons:

- 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.
- 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 behaviors, 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.

6.2 Methodology application

In the following sections, the execution of the methodology with the three different inputs (i.e., transactions, transactions internal execution, and the combination) is described. We successfully conducted the methodology on all the four smart contracts mentioned earlier and, as an example, in the following section, we will describe the execution on LANDRegistry as it is the most utilized one 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.

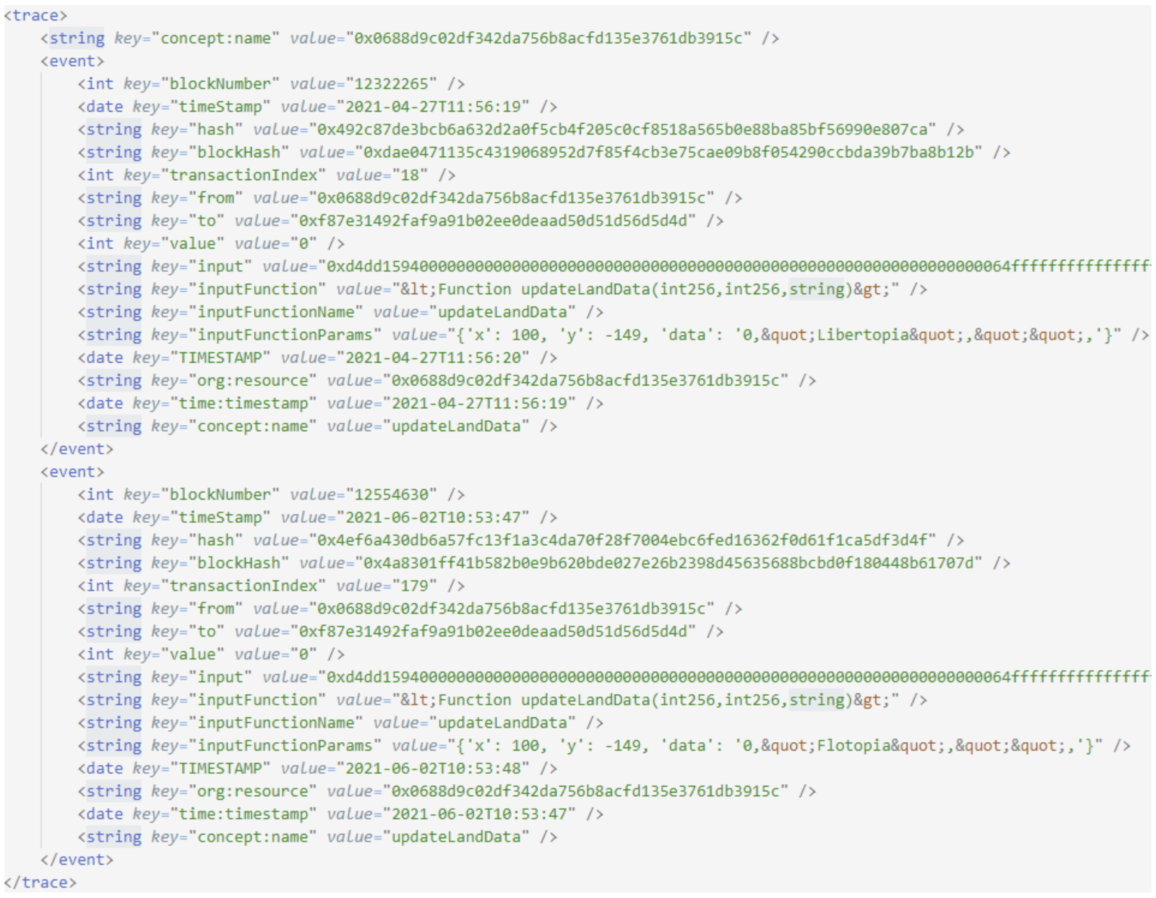
[4] `https://etherscan.io/address/0x1784ef41af86e97f8d28afe95b573a24aeda966e`
[5] `https://etherscan.io/address/0x19a8ed4860007a66805782ed7e0bed4e44fc6717`
[6] `https://ethereum.org/en/developers/docs/standards/tokens/erc-721/`
[7] `https://etherscan.io/address/0xf87e31492faf9a91b02ee0deaad50d51d56d5d4d`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

6.2.1 Transactions

The first execution is the one on transactions. We selected LANDProxy as the transaction address, and LANDRegsitry as the ABI address due to the proxy pattern. Regarding the block range we tried three different combinations:

- *14492022 - 14672860*. To monitor the methodology performance on a relatively small log: 540 traces and 1000 events.
- *12322265 - 14883354*. One-year transactions. Intermediate-size log with 4895 traces and 12299 events.
- *0 - 99999999*. Full list of transactions as of *15 April 2022* with 9184 traces and 51855 events.

For all the specified block ranges, we sorted the records by *timestamp*, selected *from* (i.e., sender) 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. A trace represents the lifecycle of transactions sent by the user. An example trace from the generated log is shown in figure 6.2.



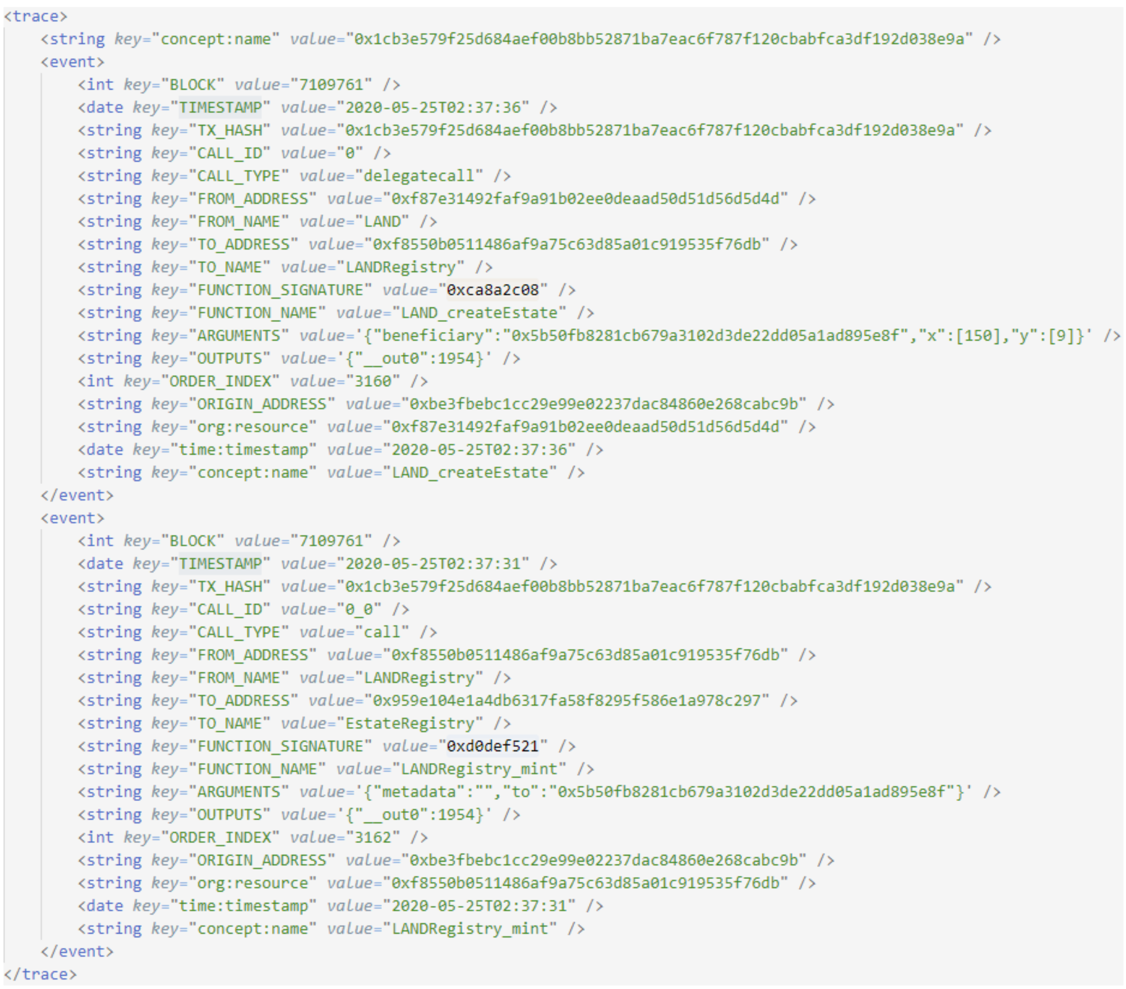
```
<trace>
    <string key="concept:name" value="0x0688d9c02df342da756b8acfd135e3761db3915c" />
    <event>
        <int key="blockNumber" value="12322265" />
        <date key="timeStamp" value="2021-04-27T11:56:19" />
        <string key="hash" value="0x492c87de3bcb6a632d2a0f5cb4f205c0cf8518a565b0e88ba85bf56990e807ca" />
        <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="0xd4dd159400000000000000000000000000000000000000000000000000000000000064ffffffffffffff
        <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="0xd4dd159400000000000000000000000000000000000000000000000000000000000064ffffffffffffff
        <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 6.2: XES trace generated from transactions

6.2.2 Transactions internal execution

In the collection of transactions internal execution for LANDProxy for the maximum block range, we were able to get 600.000 records. As described in Section 4, the application of the methodology generated 33 files, one for each function name discovered in the dataset. Each file contains the internal transactions for the transactions executing a specific function. For instance, one file for *setApprovalForAll*, another for *createEstate*, *transferFrom*, and so on. The records were sorted by timestamp, the hash of the parent transaction was selected as case ID, and the function name as *concept:name*. In this way, each trace represents the internal execution flow of a transaction. An example trace from the generated log is shown in figure 6.3.

Techniques like Process Discovery should produce models that resemble the execution flow with splits when we have decisions (e.g., if-else) and loops in correspondence of for or while. Infrequent variants or unusual branches in the model could represent bugs in the code.

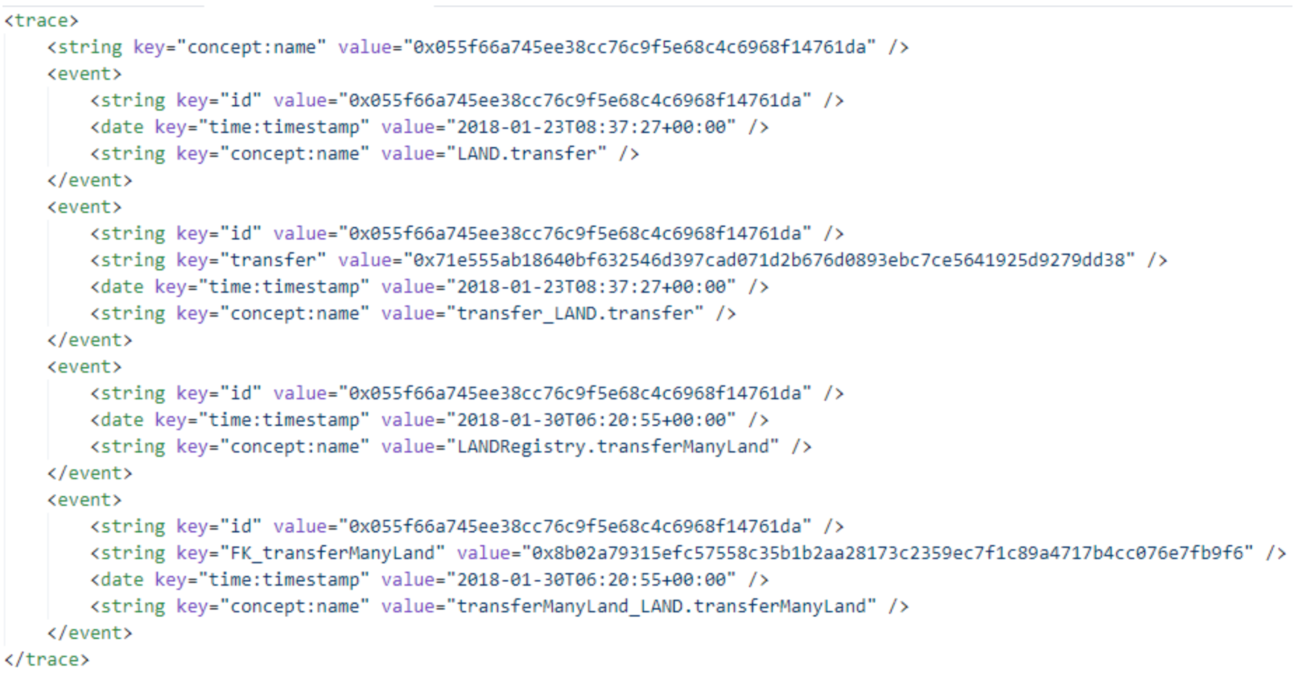


```
<trace>
    <string key="concept:name" value="0x1cb3e579f25d684aef00b8bb52871ba7eac6f787f120cbabfca3df192d038e9a" />
    <event>
        <int key="BLOCK" value="7109761" />
        <date key="TIMESTAMP" value="2020-05-25T02:37:36" />
        <string key="TX_HASH" value="0x1cb3e579f25d684aef00b8bb52871ba7eac6f787f120cbabfca3df192d038e9a" />
        <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 6.3: XES trace generated from transactions internal execution

6.2.3 Combination of transactions and internal execution

To process the combination of transactions and their internal execution, we selected a subset of the dataset with 600.000 transactions in order to avoid performance issues in the execution of Process Mining techniques. Records were sorted by timestamp, the *from* the field was selected as case ID, and the function name as *concept:name*. As described in Section 4, each function name is prefixed by the smart contract name on parent-transaction (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 6.4.



```
<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 6.4: XES trace generated for BPMN Miner

6.3 Process Mining

On the generated XES logs, we successfully applied and evaluated Process Mining techniques like Process Discovery, Conformance Checking, Simulation, and Social Network Analysis. Special attention was given to the logs generated from the combination of transactions and transactions' internal execution on which we applied BPMN Miner [5], a novel technique able to output a BPMN model with subprocesses inside.

6.3.1 Statistics

Before running Process Mining techniques, we decided to print out some statistics using PM4Py to get a better understanding of the logs. The first one was the distribution of events over time which helps to understand in which time intervals the most increased number of events is recorded. It can be shown as a chart or as a plot on different timeframes: hours, weeks, days, months, and years. On the full log, we can see that the peak of events was reached in the first months after Decentraland's launch and in the first months of 2022 when Facebook decided to rebrand to Meta putting a massive focus on metaverses. Regarding smaller timeframes, we can see that in the winter months and evening hours the transaction volume is higher.

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 visualize the distribution of cases and events over time, with the following choices: X-axis for the timestamp of the event, Y-axis for the index of the case inside the event log, and the Color for the activity of the event. The aforementioned choices permit the identification of visual patterns such as batches. In figure 6.5, we can again see the peak volume at the beginning and in correspondence of early 2022. Most of the dots represent the functions *setApprovalForAll*, *transferFrom*, and *updateLandData*. Other aspects that should look at on a dotted chart are vertical patterns. Such patterns indicate batching, i.e., shorter periods where the same activity occurs for many cases. We can see vertical patterns in the correspondence of the charts 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.

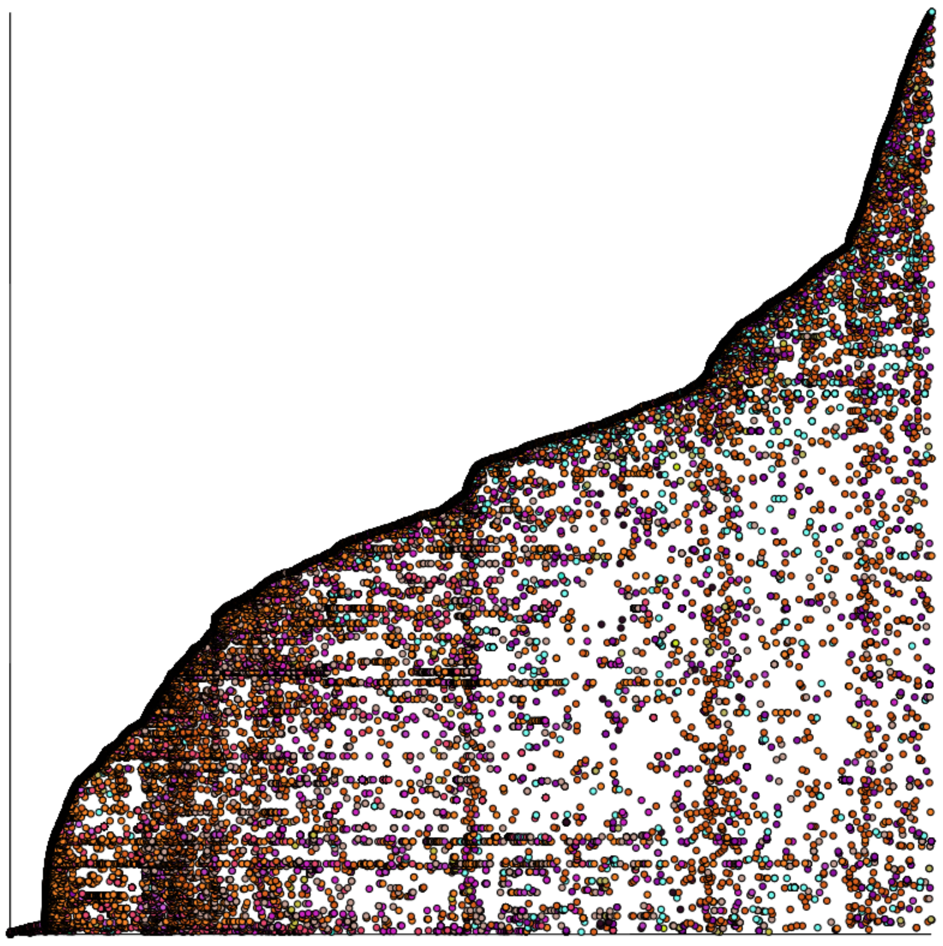


Figure 6.5: Dotted chart for the full log

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 of the operator is the Marketplace meaning that *setApprovalForAll* was called to let the latter sell the LAND. It explains also 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 extraction methodology can similarly serve multiple intents.

6.3.2 Process Discovery

The first Process Mining technique we applied to the generated logs was Process Discovery. We used PM4Py to execute the following algorithms:

- Inductive Miner (IM) [9], Inductive Miner infrequent (IMf) [11], and Inductive Miner directly-follows (IMd) [10]. 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 sublogs, 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 sublogs but uses the directly follows graph. Maximum performance but replay fitness guarantees are lost.

- Heuristic Miner [18]. 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).

- Alpha Miner [1]. The alpha miner 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.

In figures 6.8, 6.7, 6.7 are respectively shown the models discovered with Inductive Miner (IM), Inductive Miner infrequent (IMf), and Inductive Miner directly-follows (IMd) on the one-year log. The results on the full log are similar, the only difference is the lacking of deploy and setup functions executed when the smart contract were deployed. The model discovered with the Heuristic Miner is shown in figure 6.9.

To assess the discovered models we applied the metrics provided by PM4Py[9] that are obtained executing Conformance checking techniques:

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

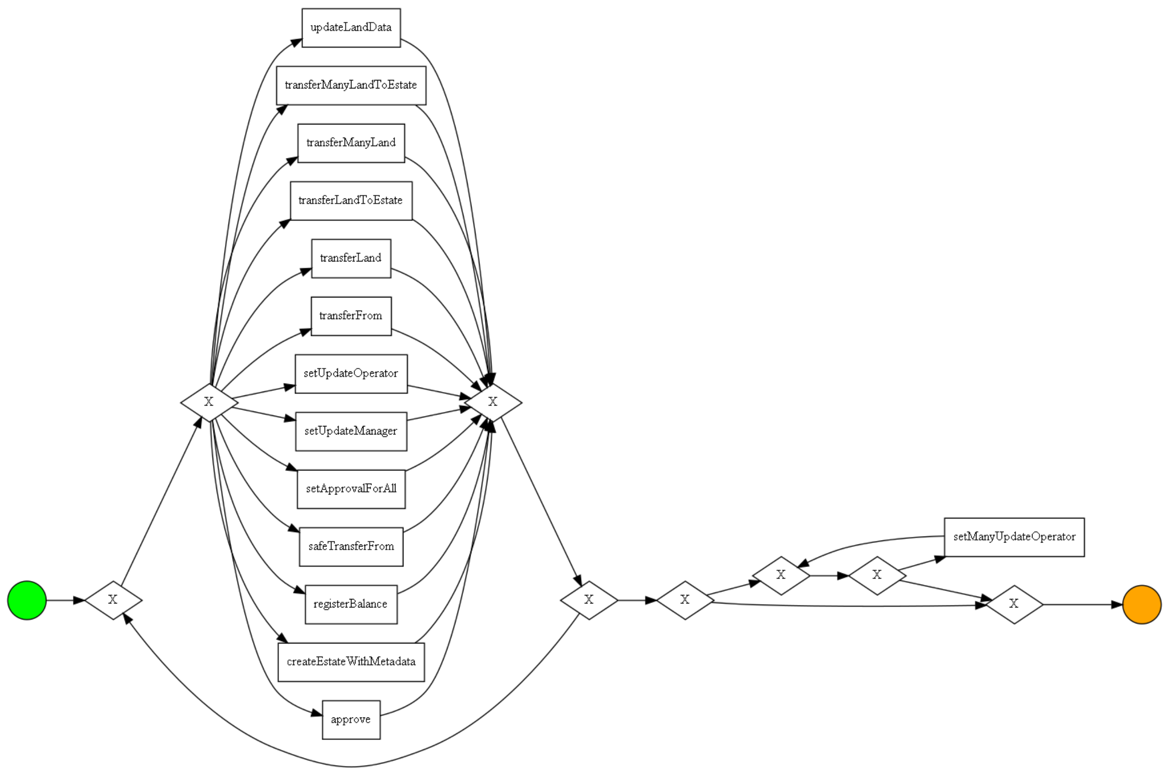


Figure 6.6: IMd model on one-year log

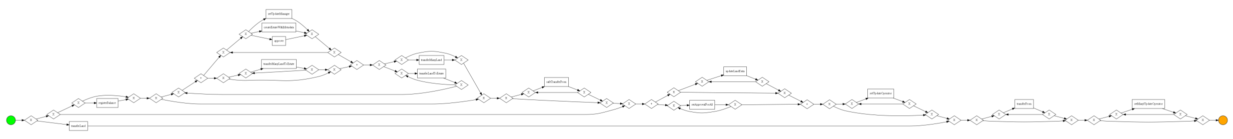


Figure 6.7: IMf model on one-year log

- 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 [2]. 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. We decided to use token-based replay because it performed better on bigger logs.

- Precision. Two approaches for the measurement of precision in PM4Py, ET-Conformance [13] which uses token-based replay and Align-ETConformance [13] 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 really meaningful. The main difference between

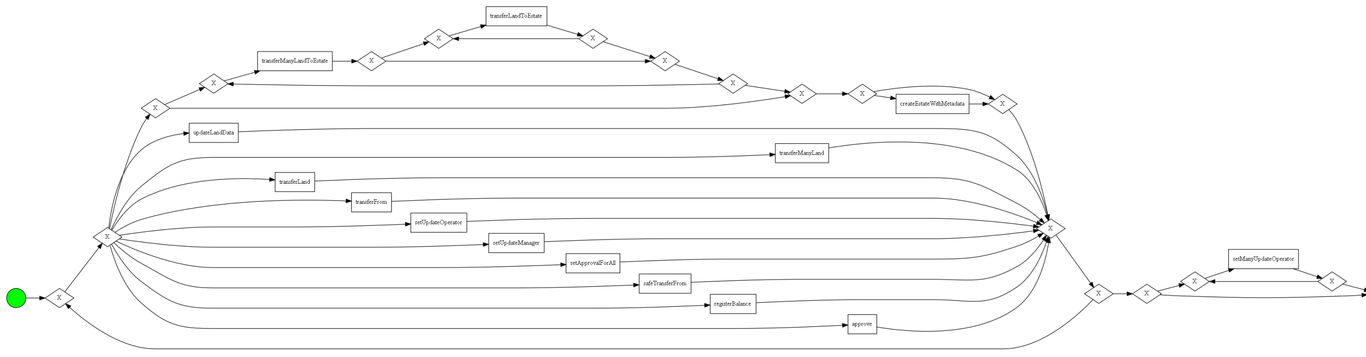


Figure 6.8: IM model on one-year log

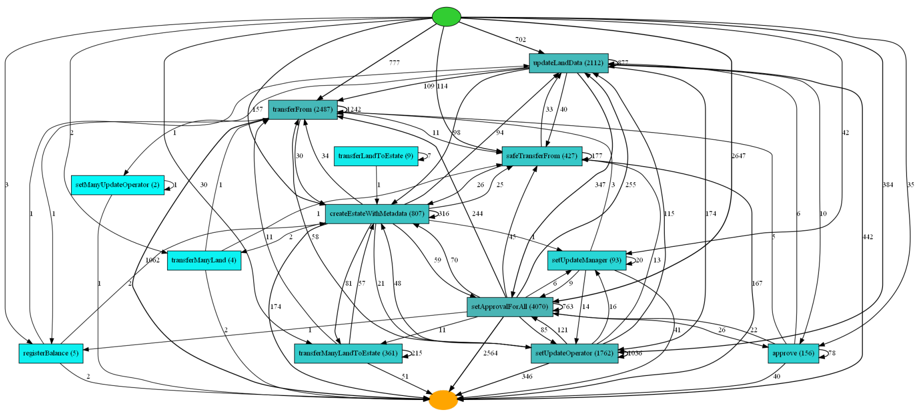


Figure 6.9: Heuristic miner model on one-year log

the approaches 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, work on any kind of relaxed sound nets, but can be slow if the state-space is huge. Also here we decided to use token-based replay because it performed better on bigger logs.

- Generalization. Analyse how the log and the process model match. PM4Py generalization measure is described in [4]. Basically, 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 (for example, 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 define simplicity taking into account only the Petri net model. The criteria that we use for simplicity is the inverse arc degree as described in [3]. First of all, we consider the average degree for a place/transition of the Petri net, that 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 $1.0/(1.0 + max(mean_degree - k, 0))$.

Algorithm	**Fitness**[2]	**Precision**[13]	**Generalization**[4]	**Simplicity**[3]
IM[9]	1	0,56	0,77	0,62
IMf[11]	0,96	0,93	0,84	0,7
IMd[10]	1	0,53	0,77	0,55
Heuristic[18]	0,98	0,84	0,7	0,46
Alpha[1]	0,56	0,8	0,8	0,47

Table 6.1: Process Discovery metrics for transactions

The properties of the Inductive Miner variants are reflected in the models. The IMd model in figure 6.6 is the most understandable but lacks of precision The Inductive Miner variants which gave the best results was the IMd. On the full, one year (shown in figure 6.6, and thousand logs, the IMd variants gave the most understandable model. IMf is the one that performed the worst since it produces a model with perfect replay fitness which is too precise and difficult to understand. IMf model is shown in figure 6.7.

6.3.3 BPMN Miner

6.4 Evaluation of results

For example, in our case study, we take the viewpoint of an individual kitty, but this may not be suitable for analysing the complete population. To generate different views, our framework allows analysts to materialize their choice of identity attributes in the manifest, and for some applications multiple manifests with different choices might be required to obtain the desired views. We mined the extracted event flows from both logs as depicted in Fig. 5. Fig. 5a shows that the developer started with two kitties initially, and bred them 3000 times for bootstrapping the game. The behavior during the everyday use (Fig. 5b) shows considerably more variation and includes all types of events. While we could delve into a deep analysis of kitty behaviour, the purpose of the case study in this paper was to test if the proposed framework can be applied to existing smart contracts – which we deem to be the case. We successfully extracted event logs, stored them in the XES format, and loaded them for analysis in both ProM and Disco.

6.5 CryptoKitties

comparison with ELF

7. Conclusions and Future Work

7.1 Conclusions

In this paper, we conducted a case study on process mining for data extracted from the blockchain application Augur. To this end, we used ELF to extract data over essentially the entire lifecycle of Augur v1.0. We used process mining methods and tools to explore the data, discover models for a set of variants, and conducted conformance checking and performance analyses. Finally, we interviewed the chief architect of Augur to validate our insights and understand their usefulness. As stated in Sect. 3, we followed open science principles and made all data and code from our study available publicly. In summary, we conclude that there is clear evidence for the usefulness of process mining on blockchain data. Main areas of interest for software developers may include user behavior analysis and security audits, for which we demonstrated the applicability of process mining tools. Indeed, we discovered a bug in Augur's smart contracts – albeit a non-critical one. Future research can be done 16 R. Hobeck, C. Klinkm¨uller, D. Bandara, I. Weber, W. van der Aalst evaluating other applications which might run on other blockchains, such as Hyperledger Fabric. The analysis method could be extended for blockchain-specific security and user studies, e.g., through drift detection and cohort analysis. [**apidoc**]

7.2 Future work

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

[3] Fabian Rojas Blum. “Metrics in process discovery”. In: 2015.

[4] 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.

[5] Raffaele Conforti et al. “BPMN Miner: Automated discovery of BPMN process models with hierarchical structure”. In: *Information Systems* 56 (2016), pp. 284–303. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2015.07.004. URL: https://www.sciencedirect.com/science/article/pii/S0306437915001325.

[6] Richard Hobeck et al. “Process Mining on Blockchain Data: A Case Study of Augur”. In: Aug. 2021, pp. 306–323. ISBN: 978-3-030-85468-3. DOI: 10.1007/978-3-030-85469-0_20.

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

[8] Christopher Klinkmüller et al. “Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications”. In: Aug. 2019, pp. 71–86. ISBN: 978-3-030-30428-7. DOI: 10.1007/978-3-030-30429-4_6.

[9] 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.

[10] 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.

[11] S.J.J. Leemans, D. Fahland, and W.M.P. Aalst, van der. “Discovering block-structured process models from event logs containing infrequent behaviour”. English. In: *Business Process Management Workshops : BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*. Ed. by N. Lohmann, M. Song, and P. Wohed. Lecture Notes in Business Information Processing. 9th International Workshop on Business Process Intelligence (BPI 2013), BPI 2013 ; Conference date: 26-08-2013 Through 26-08-2013. Germany: Springer, 2014, pp. 66–78. ISBN: 978-3-319-06256-3. DOI: 10.1007/978-3-319-06257-0_6.

[12] 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.

[13] Jorge Munoz-Gama and Josep Carmona. “A Fresh Look at Precision in Process Conformance”. In: *BPM*. 2010.

[14] Roman Mühlberger et al. “Extracting Event Logs for Process Mining from Data Stored on the Blockchain”. In: Sept. 2019.

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

[16] 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.

[17] Ingo Weber et al. “Untrusted Business Process Monitoring and Execution Using Blockchain”. In: *BPM*. 2016.

[18] 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.

Acknowledgments

Ringrazio...