Cambridge Medium (2020), **37**, 1–20

CAMBRIDGE
UNIVERSITY PRESS

ARTICLE TYPE

Event Log Extraction Methodology for Ethereum applications

Andrea Morichetta,* Yuri Paoloni, and Barbara Re

University of Camerino, Computer Science Division, Camerino, Italy
*Corresponding author. Email: andrea.morichetta@unicam.it

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, extracting logs suitable for process analysis from blockchains can help understand processes and their execution. However, extracting logs from blockchains is challenging and the techniques created so far by researchers to address this challenge have limitations. Some are tailored to specific use cases, such as blockchain-based Business Process Management Systems in which smart contracts are generated from BPMN models. Access to the models enables the use of information about the context, activities, and execution flow fundamental to the correct execution of the technique. Others focus on blockchain events, which lack a standard format and require complex processing steps to create a correlation and generate traces. To solve this challenge, the EveLogmethodology is proposed in this paper to extract logs from Ethereum smart contracts and enable the application of process analysis techniques. The proposed methodology aims to extract XES logs from public and internal blockchain transactions. Public transactions are the standard blockchain transactions that, unlike events, have standard parameters that ensure a common base in extraction operations. Internal transactions represent the inner execution flow of public transactions, which is the invocations of the functions of smart contracts involved in the process. The ability to process internal transactions is a novelty not included in existing extraction methodologies. EveLogincludes five steps: (i) extraction of data from smart contracts, (ii) decoding of raw data, (iii) selection of sorting criteria, (iv) construction of traces, and, finally, (v) generation of the XES log. The methodology has been implemented in a client-server application for end-user usage. EveLoghas been validated on CryptoKitties, a popular game developed on Ethereum. The extracted logs were used to run process discovery and perform a comparison with an existing extraction methodology.

Keywords: blockchain, ethereum, transactions, events, logs, xes, process mining, process discovery

1. Introduction

In recent years, blockchain has been acquiring attention as an innovative technology to execute business processes and regulate the interactions between the parties involved in the process. 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. These operations become appealing for process and data flow analysis techniques to better understand the processes and their actual execution. Examples are producing high-level models from process logs, comparing a process's actual execution with the intended one, analyzing business parties' interactions, etc. To get such a technique to work, it is necessary to extract information in a specific format. When dealing with processes, XES (IEEE

2016) logs are frequently employed. However, extracting XES logs from blockchains poses two major challenges:

- *The notion of "trace" is not present in the data produced by blockchains.* There isn't a built-in process identifier that groups a set of records together. Transactions are atomic actions that modify the state of the blockchains and do not have a native correlation between them. This is different from traditional systems (e.g., databases) where an identifier is typically present to group correlated records together in a trace (e.g. being part of the same process instance).
- *Timestamps are not accurate in blockchain data.* Transactions mined in the same block have the same timestamp value even though they are actually executed sequentially. The sequentiality of transactions must be represented in the output logs without relying solely on timestamps, an additional criteria is necessary.

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, Mühlberger et al. 2019 is tailored to blockchain-based Business Process Management Systems. This kind of system is generated from BPMN models. Hence, the access to models implies that the context, the activities, and the execution flow of the process under analysis are known a priori. This information isn't available for every smart contract in the blockchain. Other techniques, like Klinkmüller et al. 2019, 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 emitted in a smart contract 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, EveLogis proposed in this paper to collect data from Ethereum smart contracts and enable the application of process analysis techniques. The proposed methodology aims to extract XES logs from public[1] and internal blockchain transactions. Public transactions, unlike events, have standard parameters that ensure a common ground of operations. In every transaction, there is a guarantee that the *hash*, the *sender*, the *receiver*, and other parameters are present. The internal transactions represent the execution flow of a public transaction, which is the invocations of the functions of the smart contracts involved in the procedure. Extracting logs from internal transactions opens up new possibilities in process analysis that can be of extreme value to blockchain-based applications. The ability to process internal transactions is a novelty not included in existing extraction methodologies. The EveLogmethodology comprises five steps:

- *Data extraction*: the smart contracts addresses and the block range are selected for the data retrieval.
- *Raw data decoding*: 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.

EveLog has been validated on CryptoKitties, a popular game developed on Ethereum that allows users to breed, buy, and sell virtual cats. CryptoKitties was one of the first and most successful examples of non-fungible tokens (NFT) on the Ethereum blockchain. It was chosen for its longevity (deployed on December 2017), popularity, and, because it was used in the existing literature. The logs extracted from the CryptoKitties transactions were used to run process discovery and perform a comparison with the Ethereum Logging Framework (Klinkmüller et al. 2019) that extracts logs from blockchain events.

1. Public is used to differentiate the standard transactions from the internal transactions and avoid confusion to the reader

The rest of the paper is organised as follows. Section 2 includes a general presentation of the blockchain, Ethereum, process mining, and XES logs. Section 3 reviews relevant related works. Section 4 introduces the proposed EveLogmethodology, while Section 5 the methodology in practice. Finally, Section 6 concludes the paper by touching upon directions for future work.

2. Background

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

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. 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. The contents of the blockchain ledger are replicated across many geographically-distributed processing nodes which jointly operate the blockchain system, without the central control of any single trusted third party. A network protocol that defines rights, responsibilities, and means of communication, verification, validation, and consensus across the nodes in the network.

Ethereum. Ethereum (Buterin 2014) 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. 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. 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.

Public transactions. Public transactions, or just 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 Ethereum, a submitted transaction includes *hash*, *blockNumber*, *timestamp* (i.e., time at which the transaction has been added in a block), *to*, *from*, *value* (i.e., amount of ETH), *data* (i.e., binary code to create a smart contract, function invocation), *gasLimit*, etc.

Internal transactions. An internal transaction is the consequence of smart contract logic triggered by a blockchain transaction (i.e., regular transactions described in Section 2). A smart contract engagement can result in tens or hundreds of internal transactions which 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.

Events Events are logs issued during the execution of smart contract functions. An event hasn't standard parameters and is not mandatory present in a smart contract 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.

Process mining Process mining is a family of techniques relating to data science and process management to support the analysis of operational processes based on event logs. Process mining aims to turn event data into insights and actions. The main classes of process mining techniques are process discovery and conformance checking. In particular, 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 most common algorithms are the Alpha algorithm (Aalst, Weijters, and Măruşter 2004), the Heuristic Miner (Weijters, Aalst, and Medeiros 2006), and the Inductive Miner (Leemans, Fahland, and Aalst 2013).

XES logs The XES (IEEE 2016) standard defines a grammar for a tag-based language to provide a unified and extensible methodology for capturing event logs and event streams. The primary purpose of XES is process mining. On the top level of an XES file, 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 *events* that represent atomic granules of activity that have been observed during the execution of a process.

3. Related Works

Before beginning the research, a review of existing work on the extraction of logs from blockchains was conducted. The text parameters *"log extraction blockchain"*, *"process logs blockchain"*, and *"process mining blockchain"* was used on *Google Scholar* to search for the contributions made in recent years. To skim the obtained results, only articles that met the following criteria were analyzed: (i) describe the implementation of a log-extraction methodology, (ii) deal with the Ethereum blockchain, (iii) have a considerable number of citations. After the filtering, papers dealing with other blockchains (Beck et al. 2021) (Duchmann and Koschmider 2019) were not further analyzed. The remaining papers were: *"Extracting Event Logs for Process Mining from Data Stored on the Blockchain"* (Mühlberger et al. 2019), *"Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications"* (Klinkmüller et al. 2019), and *"Process Mining on Blockchain Data: A Case Study of Augur"* (Weber et al. 2021). The examination of the mentioned works highlighted limitations and room for additional research. The table 1 presents an overview of the analyzed papers with the *methodology*, the *target data*, the *target application*, the *case ID*, and the *process discovery algorithm*.

Table 1. Related work overview

Methodology	Data	Target	Case ID	Discovery algorithm
BPMS[a]	Transaction	Incident-management	transaction.to	Inductive Visual Miner
ELF[b]	Event	CryptoKitties	Kitty ID	DFGs
ELF[c]	Event	Augur	Market ID	Inductive Miner

a Mühlberger et al. 2019
b Klinkmüller et al. 2019
c Weber et al. 2021

3.1 Process mining on blockchain-based BPMS

Mühlberger et al. 2019 defined a methodology to extract logs from the transactions involving smart contracts generated with a blockchain-based Business Process Management System (BPMS) like Caterpillar (López-Pintado et al. 2019) and Lorikeet (Tran, Lu, and Weber 2018). This kind of system is able to translate BPMN or Choreography models into smart contracts that 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. This information is not available for 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. Figure 1 shows the proposed approach from extraction to transformation to XES logs.

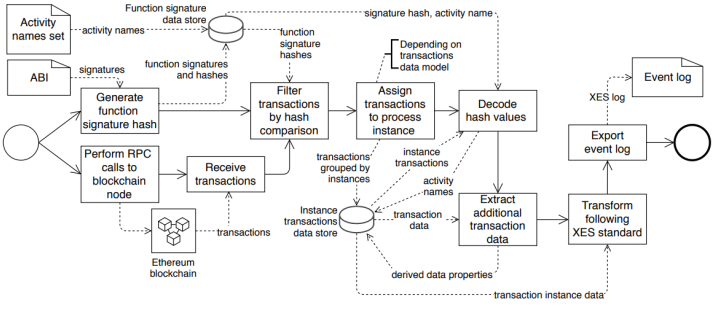

Figure 1. Overview of the methodology proposed in Mühlberger et al. 2019

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 2. The function selector is obtained by computing the KECCAK[2] 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.

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 2. Activity dictionary

2. https://keccak.team/index.html

The authors successfully evaluated the methodology on an Incident-management process (Weber et al. 2016) generating an XES log and applying on 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 foreseen by the authors who decided to focus specifically on blockchain-based BPMS.

3.2 Ethereum Logging Framework

The other analyzed papers are *"Mining Blockchain Processes: Extracting Process Mining Data from Blockchain Applications"* (Klinkmüller et al. 2019) which describes the implementation of the Ethereum Logging Framework (ELF) and *"Process Mining on Blockchain Data: A Case Study of Augur"* (Weber et al. 2021) which illustrates in detail a case study of ELF. Klinkmüller et al. 2019 proposed a framework to extract process event data from applications that utilize the Ethereum's transaction log (i.e., Solidity events[3]). The framework comprises three main parts: (i) a *manifest* to specify the target smart contract addresses, the block range, and transformation rules (i.e., mappings from Ethereum logs to XES data), (ii) an *extractor* that apply the manifest rules; (iii) a *generator* that generates Solidity code to emit events in line with the manifest specifications.

To extract blockchain event data, it is necessary to provide the 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 in the manifest 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 Section 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 mappings. 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., discovered 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. 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[4] (Klinkmüller et al. 2019) and Augur[5] (Weber et al. 2021). 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.

4. EveLogMethodology

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 without limitations. An extraction methodology capable of collecting data from every Ethereum smart contract and generating XES logs ready for process analysis without requiring complex

3. https://docs.soliditylang.org/en/v0.8.16/contracts.html#events
4. https://www.cryptokitties.co/
5. https://augur.net/

configurations. To work with every Ethereum smart contract, the EveLogmethodology 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 for EveLog. Thanks to transactions, the methodology can rely on the *hash*, *from*, *to*, and other transaction fields being always present in the collected data.

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

- **public transactions**: collect public transactions from smart contracts and generate an XES log according to the selected parameter: *sort by*, *case ID*, and *concept name*.
- **internal transactions**: collect internal transactions from 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 public and internal transactions**: combine public transactions and the respective internal transactions to obtain logs that contain sub-processes. Such logs can be exploited with techniques that are able to compute processes and sub-processes.

4.1 Steps of the Methodology

The EveLogmethodology, as depicted in the figure 3, consists of five steps described in the following sections: (i) *data extraction*, (ii) *raw data decoding*, (iii) *sorting criteria selection*, (iv) *trace construction*, and (v) *XES generation*.

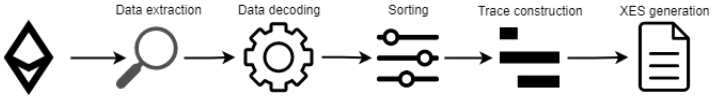

Figure 3. Overview of the methodology

4.1.1 Data Extraction

In the *data extraction* step, as the name implies, blockchain data is extracted according to two parameters: *smart contract addresses* and *block range*. The first parameter indicates the addresses from which to collect transaction data. The second parameter is optional and is used to scope the research in a precise block number interval. More specifically, EveLogrequires 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)[6] 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[7] to allow the upgradability of the smart contract. A proxy pattern 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 proxy contract and use the ABI of the main contract to decode the hexadecimal fields.

Public transactions The collected public transactions are in the format described in Section 2. Public transactions can be collected using blockchain explorers like Etherscan [8]. The Algorithm 1 shows the code for public transactions collection.

6. https://docs.soliditylang.org/en/v0.8.13/abi-spec.html
7. https://blog.openzeppelin.com/proxy-patterns/
8. https://etherscan.io/

Algorithm 1 Public transactions retrieval

Require: *contracts_addresses*, *start_block*, *end_block*
1: **for all** *address* in *contracts_addresses* **do**
2: *public_transactions* = *etherscan.get_txs_by_address*(*address*, *start_block*, *end_block*)
3: **end for**

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 public transaction execution. To collect such data, specific tools are needed. EthTx[9], an advanced decoder for blockchain transactions, allows to collect internal transactions distinguished by the *call_id* field. This field permits differentiating a public transaction from an internal transaction. Example are: *"\\n"* (public transaction), *"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. The algorithm to collect internal transactions is similar to Algorithm 1 with the only difference of EthTx instead of Etherscan for data collection.

4.1.2 Raw data decoding

The collected raw data requires a decoding step. Some fields of the collected public transactions are encoded in hexadecimal according to the smart contract ABI. One of the encoded 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 *data extraction* step. Then, the algorithm iterates through the public transactions to decode the *input* field and store the function name and parameters in new fields (i.e., *function*, *function_params*) (lines 2-6). In addition, fields not needed in the next steps are removed, e.g., *nonce*, *value*, *isError*, *input*, etc. (line 7).

Algorithm 2 Data decoding

1: *contractABI* = *get_contract_ABI*(*contract*)
2: **for all** *transaction* in *public_transactions* **do**
3: *function*, *function_params* = *decode_input*(*contract_ABI*, *transaction.input*)
4: *transaction.function* = *function*
5: *transaction.function_params* = *function_params*
6: **end for**
7: *transactions.remove*([“*nonce*”, “*value*”, “*isError*”, “*input*”...])

The decoding step is not performed on internal transactions since the data obtained with *EthTx* contains both the raw and the decoded fields. Instead, unnecessary fields are removed also there: *load_id*, *chain_id*, *raw_arguments*, *raw_outputs*, *error*, etc.

4.1.3 Sorting criteria selection

The sorting step is common between public and internal transactions. In this step, one or more fields from the 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 Section 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 public transaction has the *transactionIndex* field that indicates the ordering of a transaction inside a block. This reflects also the execution order of

9. https://ethtx.info/

public transactions in a block[10]. 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.

4.1.4 Trace construction

The *trace construction* step 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 mentioned techniques and allows to highlight different aspects. The trace construction algorithm differs for the three data inputs supported by the EveLogmethodology.

Public transactions For the public transaction trace construction phase, only the trace identifier (i.e. *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 instance, if the field *from* (i.e., the sender of the transaction) is selected as *case ID*, all the public transactions sent by an address will be included in a trace. Performing process discovery on such a log will discover a model representing the lifecycle of public transactions sent by that address to a specific contract.

Algorithm 3 Public transactions trace construction

1: *public_transactions.case_concept_name = public_transactions.from*
2: *public_transactions.time_timestamp = public_transactions.timeStamp*
3: *public_transactions.concept_name = public_transactions.inputFunctionName*

As shown in Algorithm 3, there isn't a complex manipulation of the records to generate traces. In addition to the *case ID*, other fields related to the *Concept*[11] and *Time*[12] extensions are set:

- **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.

As an example, in the Algorithm 3, the field *from* is selected as *case:concept:name* (i.e., *case ID*) to generate traces containing all the public transactions sent by an account (line 1). The field *timeStamp* is selected as *time:timestamp* (line 2). The field *inputFunctionName* is selected as *concept:name* (line 3) to name the XES event with the smart contract function executed in the public transaction.

Internal transactions For internal transactions, the situation is more complex, and a dedicated algorithm, shown in Algorithm 4, is needed. As mentioned at the beginning of the section, 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 in the execution of a public transaction. Each of the created logs will contain traces of internal transactions that refer to a specific function (i.e. *function name*) performed in a public transaction. The data obtained with EthTx contains both public and internal transactions. Each of the records has a *tx_hash* field that indicates the public transaction to which it belongs. This field is especially useful to understand which public transaction an internal transaction is a part of. In each created log, the traces will represent the internal executions of the

10. https://ethereum.org/en/developers/docs/apis/json-rpc/
11. http://www.xes-standard.org/concept.xesext
12. http://www.xes-standard.org/concept.xesext

public transactions, therefore, internal transactions should be grouped into traces using the public transaction hash. To create different logs for each *function name* and distinguish public transactions from internal transactions, the *call_id* field can be used (described in 4.1.1). Records with *call_id* == *"\\n"* refer to public transactions and the records with *call_id !*= *"\\n"* to internal transactions. The algorithm iterates through the list of public and internal transactions collected with EthTx (lines 4–16) and stores, in correspondence of *call_id* == *"\\n"*, the sender address in the variable *user_address* (line 6) and the function name in a new field named *new_hash _group* (line 7). The *new_hash _group* field will have the value of the *function name* for public transactions and a *null* value for internal transactions. This is ideal for the *"group by"* operation that will be executed in line 13. In the same iteration, the algorithm stores on each record the value of the variable *user_address* (i.e., sender address) in the field *origin_address* (line 11) since in internal transactions the *from_address* field refer to the previous smart contract in the execution trace and not to the sender of the public transaction. This is done to keep track of the original sender in the records of internal transactions. The algorithm proceeds by prefixing the *function name* of internal transactions with the name of the smart contract executing them, i.e. *contract_name.function_name* (line 9). This occurs to identify the smart contract that performs a function, as it may happen that function names are similar among smart contracts of the same application. The last step is the splitting of records into different sets where each contains the internal transactions for a specific *function name* (lines 13–16). This happens by performing a *group by tx_hash* and mapping, again, *tx_hash* to the first value of the *new_hash _group* column (line 13) as there can only be one *call_id* == *"\\n"* per *tx_hash*. Finally, each group is stored in a different data structure (lines 14–16).

Algorithm 4 Internal transactions trace construction

```
1:  user_address = " "
2:  transactions = [...]
3:  function_names = [ ]
4:  for all (transaction, index) in transactions do
5:      if transaction.call_id = "\\n" then
6:          user_address = transaction.from_address
7:          transactions.new_hash_group = transaction.function_name
8:      else
9:          transactions[index].function_name =
                "transaction.from_name + "_" + transaction.function_name"
10:     end if
11:     transactions[index].origin_address = user_address
12: end for
13: transactions.group = transactions.tx_hash.map(
        transactions.groupby("tx_hash").new_hash_group.first())
14: for all group in transactions.groupby("group") do
15:     function_names.push(group)
16: end for
```

Combination of public and internal transactions The procedure for the combination of public and internal transactions is shown in Algorithm 5. As said at the start of the section, the end goal is to generate a base format for XES logs that can be exploited by algorithms that deal with processes and sub-processes. The generated logs can then be adapted to suit the needs of a specific algorithm as later will be done in Section 5 with BPMN Miner (Conforti et al. 2014).

The input is the dataset collected with EthTx that contains both public (i.e., call_id = "\\n") and internal transactions (i.e., call_id != "\\n"). The strategy is to add on the record of internal transactions a new field with the key equal to the name of the function executed in a public transaction and the value equal to the hash of the public transaction (line 13) (e.g., *"transfer": "0x..."*). In this way, a

relationship is created between internal transactions that are part of the same function execution. Moreover, the *function_name* of the records with *call_id = "\\n"* is prefixed with the smart contract name (i.e., *to_name*) (line 9), and the *function_name* of records with *call_id != "\\n"* is prefixed with *{function_name}_{from_name}* (line 12) to highlight the sub-process name (i.e., *function_name)* and the smart contract that is calling it (i.e., *from_name*). The *id* key is added to public and internal transactions to group them in traces (line 15). The value of *id* changes based on what is selected to generate traces. In the algorithm, the address of the user invoking the public transactions (i.e. *user_address)* is selected to generate traces containing the list of public transactions, and respective internal transactions, sent by the user. 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 13) to identify sub-processes.

Algorithm 5 Public and internal transactions trace construction

```
1:  user_address = " "
2:  function_name = " "
3:  calls_length = " "
4:  for all (transaction, index) in transactions do
5:      if transaction.call_id = "\\n" then
6:          calls_length = 0
7:          user_address = transaction.from_address
8:          function_name = transaction.function_name
9:          transactions[index].function_name = transaction.to_name + "." + transaction.function_name
10:     else
11:         calls_length+ = 1
12:         transactions[index].function_name = function_name + "_"
                +transaction.from_name + "." + transaction.function_name
13:         transactions[index][function_name] = transaction.tx_hash
14:     end if
15:     transactions[index].id = user_address
16: end for
```

4.1.5 XES log generation

After the trace construction step, the EveLogmethodology is finally ready to generate the XES log according to the parameters selected in the previous steps. The generated XES logs contain two additional extensions:

- *Concept*[13] extension: defines, for all levels of the XES type hierarchy, an attribute that stores the generally understood name of type hierarchy elements.
- *Time*[14] extension: defines, for events, an attribute that stores the precise timestamp at which the event occurred.

4.2 Implementation of the methodology

The EveLogpublic transactions step of the methodology has been implemented with Python and integrated into a client-server application[15] for end users' usage. The steps involving internal transactions have not been included because the fetching requires a complete Ethereum node with the debug option for which public access could not be given. The application allows the collection of public transactions and the generation of XES logs given one or more smart contracts and a block range. The client-server application source code is publicly available in the project's GitHub

13. http://www.xes-standard.org/concept.xesext
14. http://www.xes-standard.org/time.xesext
15. https://xes-ethereum-extractor.herokuapp.com/

repository[16]. The repository contains also the Python scripts, Jupyter notebooks[17], datasets, and models used during the implementation.

Figure 4 illustrates the structure of the client-server application. In the client, the *Next.js*[18] framework is used to render the content and makes requests to the server. The requests on the server are handled by *Flask*[19], which will then route them to the proper service, e.g., *Pandas*[20] to process data, *Etherscan* to fetch transactions, and *PM4Py*[21] to build traces and generate the XES logs.

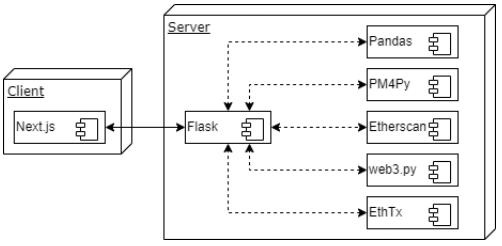

Figure 4. Client-server application

The application consists of three pages: the landing page, the tool page, and the about page with details on how to fill out forms, generate data, and more. The tool page includes two steps: the transaction collection and the XES log generation. In the first step, the user specifies the smart contracts from 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 specified in the code. 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.

The second step (i.e. XES log generation) utilizes the transaction collected in the previous step and requires the following parameters:

- *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.

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. The generated XES file can be downloaded at the end of the process.

16. https://github.com/yuripaoloni/xes-ethereum-extractor
17. https://jupyter.org/
18. https://nextjs.org/
19. https://flask.palletsprojects.com/en/2.2.x/
20. https://pandas.pydata.org/
21. https://pm4py.fit.fraunhofer.de/

5. Methodology in practice

In order to demonstrate the validity of the methodology, EveLogwas executed on CryptoKitties, a popular Ethereum game that allows users to breed, buy, and sell virtual cats. The logs extracted from the CryptoKitties transactions were used to run process discovery and perform a comparison with the Ethereum Logging Framework (Klinkmüller et al. 2019) that extracts logs from blockchain events as described in Section 3.

5.1 Case study: CryptoKitties

CryptoKitties is a decentralized application (DApp) built on the Ethereum blockchain that allows users to breed, buy, and sell virtual cats called cryptoKitties. CryptoKitties are unique digital assets that are stored on the Ethereum blockchain and each of them has a unique set of attributes, or "genes", that determine its appearance and characteristics. Users can breed two cryptoKitties together to create a new cryptoKitty with a combination of the parent's attributes. Users can also buy and sell cryptoKitties using Ether, the Ethereum cryptocurrency. The price of a cryptoKitty depends on its attributes and rarity. Some cryptoKitties have sold for hundreds of thousands of dollars. CryptoKitties was one of the first and most successful examples of a non-fungible token (NFT) on the Ethereum blockchain. IT was chosen for its longevity (deployed on December 2017), popularity, and, because it was used in the existing literature.

5.2 Working with the methodology

The following describes the execution of EveLogon CryptoKitties with *SaleClockAuction*[22] and *KittyCore*[23], together, as input smart contracts, and *6605101 - 6618097* as block range. Such configurations are the same used in Klinkmüller et al. 2019 and have been selected to perform a comparison. The *data extraction*, *raw data cleaning*, and *sorting* steps of EveLogare similar for public transactions, internal transactions, and the combination while the *trace construction* step follows a different algorithm for each data input. On the generated logs, the Heuristic Miner (Weijters, Aalst, and Medeiros 2006) has been executed with the default PM4Py's parameters: *dependency_threshold at 0.5*, *and_threshold at 0.65*, and *loop_two_threshold at 0.5*.

5.2.1 Public transactions

In this section, a comparison with the Ethereum Logging Framework (ELF) (Klinkmüller et al. 2019) is performed since public transactions are the input data of the framework. 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. The *id* of a cryptoKitty has been selected as the process instance in both cases. The lifecycle of a kitty is a trace. The extraction process with the selected configurations generated a log with *545* traces and *8936* events.

The figure 5 shows, on the left, the model discovered from the log generated with EveLogand, on the right, the model discovered from the log generated with ELF.

22. https://etherscan.io/address/0xb1690c08e213a35ed9bab7b318de14420fb57d8c
23. https://etherscan.io/address/0x06012c8cf97bead5deae237070f9587f8e7a266d

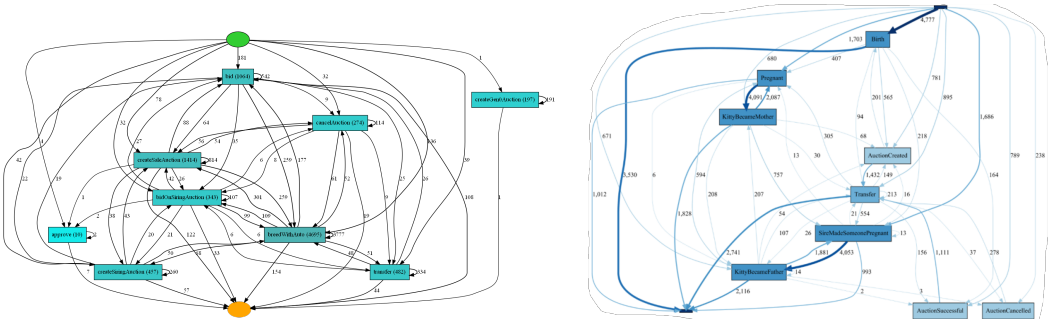

Figure 5. Left: Heuristic Miner on CryptoKitties log generated with EveLog. Right: Heuristic Miner on CryptoKitties ELF-generated log

The first difference is in the name and number of activities present in the generated logs and discovered models.

- ELF: *AuctionCreated, AuctionSuccessful, AuctionCancelled, Birth, GiveBirht, Pregnant, KittybecameMother, Transfer, SireMadeSomeonePregnant, KittyBecameFather.*
- EveLog: *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. Klinkmüller et al. 2019 mapped the blockchain event *Birth* to the XES events *Birth, GiveBirth, KittyBecameMother, KittyBecameFather*, and the blockchain event *Pregnant* to the XES events *Pregnant, SireMadeSomeonePregnant*. This justifies the difference in the number of activities between the two methodologies. 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 blockchain 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 EveLogmethodology automatically catches two different XES events, while ELF needed the mapping on *Pregnant*. This exhibits that extracting logs from blockchain events requires 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 XES events (e.g., *Birth* is surely clearer than *breedWithAuto*) and could be an improvement that can be implemented in EveLog. By introducing mappings, EveLogwill also be able to catch functions that emit more than one event. Further information on future works and possible improvements in Section 6.

5.2.2 Internal transactions

As described in Section 4, the *trace construction* step on internal transactions should generate a log for each function name discovered in the dataset. *4* logs were generated using the internal transactions collected using the configurations specified above. Each log contains the internal transactions of the public transactions executing a specific function. The functions in the log involving internal

transactions were *bidOnSiringAuction*, *createGen0Auction*, *createSaleAuction*, and *createSiringAuction*. The *hash* of the public 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 public transaction. An example trace from the generated logs is shown in listing 1. The *case:concept:name* of the trace (i.e. *case ID*) is equal to the *tx_hash* attribute in the events belonging to the trace (lines 2–3). The events have the *concept:name* attribute (line 21) equal to the *function_name* attribute (line 15) that was prefixed with the value of the *from_name* attribute (line 11).

The execution of the Heuristic Miner on such logs produces models that resemble the execution flow of a smart contract function with splits in correspondence of decisions constructs (e.g., *if, else*) and loops in correspondence of *for* or *while* constructs. Infrequent variants or unusual branches in the model could indicate bugs in the code.

```
1  <trace>
2      <string key="case:concept:name"
3      value="0x233fa05e1465b675efc20abab3f9331eab51907b3e89a4eb60a9ed217cd73321" />
4      <event>
5          <int key="block" value="6605106" />
6          <date key="timestamp" value="2018-10-29T11:48:26+00:00" />
7          <string key="tx_hash" value="0
       x233fa05e1465b675efc20abab3f9331eab51907b3e89a4eb60a9ed217cd73321" />
8          <string key="call_id" value="0" />
9          <string key="call_type" value="call" />
10         <string key="from_address" value="0x06012c8cf97bead5deae237070f9587f8e7a266d" />
11         <string key="from_name" value="KittyCore" />
12         <string key="to_address" value="0xb1690c08e213a35ed9bab7b318de14420fb57d8c" />
13         <string key="to_name" value="SaleClockAuction" />
14         <string key="function_signature" value="0x27ebe40a" />
15         <string key="function_name" value="createAuction" />
16         <string key="arguments" value='{"_duration": 2592000, "_endingPrice":
       197000000000000000, "_seller": "0x8949db9fbb4716ce5a2803085c7732c14fe03a37", "
       _startingPrice": 197000000000000000, "_tokenId": 1135862&#10;}' />
17         <string key="outputs" value="{}" />
18         <int key="order_index" value="2088" />
19         <string key="origin_address" value="0x8949db9fbb4716ce5a2803085c7732c14fe03a37" /
       >
20         <date key="time:timestamp" value="2018-10-29T11:48:26+00:00" />
21         <string key="concept:name" value="createAuction" />
22     </event>
23     ...
24 </trace>
```

Listing 1. XES trace generated from internal transactions

The figure 6 presents the model for the function *bidOnSiringAuction*. The corresponding Solidity implementation can be viewed in listing 2. The model depicts that the initial internal transaction is *getCurrentPrice*. Afterward, the model splits into two paths: (i) the execution terminates directly, or (ii) continues with the *bid* function. This behavior is also evident in the code (lines 10 and 11) through the use of a *require* control statement, which can halt the function's execution. Going ahead, the functions *bid* (line 13) and *_breedWith* (line 14) are executed. It is important to note that the *_breedWith* function is not depicted in the model as it has the *internal* modifier. An internal function is executed within the same contract and does not result in an internal transaction. However, the execution of *_breedWith* leads to the subsequent execution of *fallback* and *transfer*. The implementation for these functions can be found on Etherscan[24].

```
1  function bidOnSiringAuction(uint256 _sireId, uint256 _matronId)
2          external
3          payable
4          whenNotPaused
5      {
6          require(_owns(msg.sender, _matronId));
7          require(isReadyToBreed(_matronId));
8          require(_canBreedWithViaAuction(_matronId, _sireId));
```

24. https://etherscan.io/address/0x06012c8cf97bead5deae237070f9587f8e7a266d#code

```
9
10          uint256 currentPrice = siringAuction.getCurrentPrice(_sireId);
11          require(msg.value >= currentPrice + autoBirthFee);
12
13          siringAuction.bid.value(msg.value - autoBirthFee)(_sireId);
14          _breedWith(uint32(_matronId), uint32(_sireId));
15      }
```

Listing 2. bidOnSiringAuction function Solidity implementation

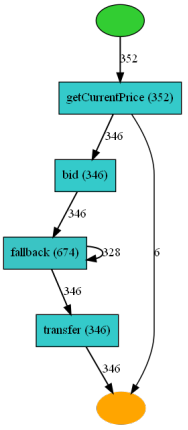

Figure 6. Model discovered from the internal transactions of *bidOnSiringAuction*

5.2.3 Combination: trace construction

As described, in Section 4, the combination of public and internal transactions requires an algorithm able to deal with the notion of processes and subprocesses. To that extent, BPMN Miner (Conforti et al. 2014) was selected to exploit the combination of public and internal transactions. It is an algorithm able to output a BPMN model with sub-processes, boundary events, and activity markers inside. 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 Conforti et al. 2014.

In order to apply BPMN Miner on the generated log, some changes to the records are required. BPMN Miner allows the selection of the keys for which sub-processes are wanted. To make it easier to select the correct events, the *function_name* for events belonging to execution traces with length one (i.e., without internal transactions) was prefixed with _ to indicate that they should not be selected. These modifications have been done using as reference the artificial log contained in the BPMN Miner source code[25]. The *from* 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 transactions (e.g., *KittyCore.breedWithAuto*) and by *{function_name}_{from}* on internal transactions (e.g., *bidOnSiringAuction_KittyCore.getCurrentPrice*). An example trace from the generated log is shown in listing 3. The *case:concept:name* of the trace (i.e. *case ID*) is equal to the *id* attribute in the events belonging to the trace (line 2). The *concept:name* of the events representing parent transactions is obtained by prefixing the smart contract name to the *function name* (e.g., *KittyCore.breedWithAuto*) (line 6). Instead, the *concept:name* of events representing internal transactions is obtained by prefixing also the *function name* of the parent transaction (e.g., *bidOnSiringAuction_KittyCore.getCurrentPrice*)

25. https://svn.win.tue.nl/repos/prom/Packages/BPMNMiner/Trunk/log/

to mark the parent process to which the event belongs (line 12). The internal transactions events contain an attribute with _ as the prefix to indicate that they're internal transactions during the BPMN Miner execution. The trace construction step generated a log with *476* traces, *13946* events, and *4* different event types.

```
<trace>
    <string key="case:concept:name" value="0x8949db9fbb4716ce5a2803085c7732c14fe03a37" />
    <event>
      <string key="id" value="0x8949db9fbb4716ce5a2803085c7732c14fe03a37" />
      <date key="time:timestamp" value="2018-10-29T11:48:26+00:00" />
      <string key="concept:name" value="KittyCore.createSaleAuction" />
    </event>
    <event>
      <string key="id" value="0x8949db9fbb4716ce5a2803085c7732c14fe03a37" />
      <string key="createSaleAuction" value="0
     x233fa05e1465b675efc20abab3f9331eab51907b3e89a4eb60a9ed217cd73321" />
      <date key="time:timestamp" value="2018-10-29T11:48:26+00:00" />
      <string key="concept:name" value="createSaleAuction_KittyCore.createAuction" />
    </event>
    <event>
      <string key="id" value="0x8949db9fbb4716ce5a2803085c7732c14fe03a37" />
      <string key="createSaleAuction" value="0
     x233fa05e1465b675efc20abab3f9331eab51907b3e89a4eb60a9ed217cd73321" />
      <date key="time:timestamp" value="2018-10-29T11:48:26+00:00" />
      <string key="concept:name" value="createSaleAuction_SaleClockAuction.transferFrom"
     />
    </event>
    <event>
      <string key="id" value="0x8949db9fbb4716ce5a2803085c7732c14fe03a37" />
      <date key="time:timestamp" value="2018-10-29T11:54:38+00:00" />
      <string key="concept:name" value="KittyCore.createSaleAuction" />
    </event>
        ...
</trace>
```

Listing 3. XES trace generated for BPMN Miner

6. Conclusion and future works

In this paper, a new extraction methodology was proposed to facilitate the task of extracting XES logs from blockchain data. The main feature of EveLogis 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 execution on CryptoKitties demonstrates the applicability of the proposed methodology in a real-world application. Process discovery was applied successfully to the generated logs but the list of applicable techniques is not limited to it: others, such as conformance checking, social network analysis, usage profiling, and simulations, can also be applied to the logs. Extracting XES logs from internal transactions is something that, to our knowledge, has never been done before. Internal transactions are difficult to retrieve because they are not stored on-chain and few tools support their collection. However, extracting logs from internal transactions allows inspecting the execution trace of immutable applications, such as smart contracts, which can help discover bugs and prevent exploits, especially in the context of blockchain, where assets (e.g., cryptocurrency, decentralized finance) and, in general, values are managed.

While we are pleased with the achieved results, there is still potential for further research.

- *Validate EveLog for multiple use cases*: we ran the EveLogon CryptoKitties to demonstrate its feasibility. The methodology should be validated on more use cases from different sectors, such as healthcare, manufacturing, the internet of things, decentralized finance, metaverses, and blockchain-based games.
- *Continue the validation with existing process discovery algorithms*: we limited the application of process discovery algorithms to the Heuristic Miner. 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 EveLog*: we conducted the research to demonstrate the applicability of the proposed extraction methodology. From now on, we can focus on the implementation of additional features to improve the usability and performance of EveLog. For instance, the mapping of the fields in the log to one or more values similar to what has been done on ELF (Klinkmüller et al. 2019) can enhance the expressivity of the fields and permit complex manipulations.

References

Aalst, Wil, A. Weijters, and Laura Măruşter. 2004. Workflow mining: discovering process models from event logs. *Knowledge and Data Engineering, IEEE Transactions on* 16 (October): 1128–1142. https://doi.org/10.1109/TKDE.2004.47.

Beck, Paul, Hendrik Bockrath, Tom Knoche, Mykola Digtiar, Tobias Petrich, Daniil Romanchenko, Richard Hobeck, Luise Pufahl, Christopher Klinkmüller, and Ingo Weber. 2021. Blf: a blockchain logging framework for mining blockchain data (September).

Buterin, Vitalik. 2014. *Ethereum: a next-generation smart contract and decentralized application platform.* https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf. [Online; accessed 16-September-2022].

Conforti, Raffaele, Marlon Dumas, Luciano García-Bañuelos, and Marcello La Rosa. 2014. Beyond tasks and gateways: discovering bpmn models with subprocesses, boundary events and activity markers. In *Business process management,* edited by Shazia Sadiq, Pnina Soffer, and Hagen Völzer, 101–117. Cham: Springer International Publishing. ISBN: 978-3-319-10172-9.

Duchmann, Frank, and Agnes Koschmider. 2019. Validation of smart contracts using process mining.

IEEE. 2016. Ieee standard for extensible event stream (xes) for achieving interoperability in event logs and event streams. *IEEE Std 1849-2016.*

Klinkmüller, Christopher, Alexander Ponomarev, An Binh Tran, Ingo Weber, and Wil van der Aalst. 2019. Mining blockchain processes: extracting process mining data from blockchain applications. In *Business process management: blockchain and central and eastern europe forum,* edited by Claudio Di Ciccio, Renata Gabryelczyk, Luciano García-Bañuelos, Tomislav Hernaus, Rick Hull, Mojca Indihar Štemberger, Andrea Kő, and Mark Staples, 71–86. Cham: Springer International Publishing.

Leemans, Sander J. J., Dirk Fahland, and Wil M. P. van der Aalst. 2013. Discovering block-structured process models from event logs - a constructive approach. In *Application and theory of petri nets and concurrency,* edited by José-Manuel Colom and Jörg Desel, 311–329. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-38697-8.

López-Pintado, Orlenys, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. 2019. Caterpillar: a business process execution engine on the ethereum blockchain. *Software: Practice and Experience* 49:1162–1193.

Mühlberger, Roman, Stefan Bachhofner, Claudio Di Ciccio, Luciano García-Bañuelos, and Orlenys López-Pintado. 2019. Extracting event logs for process mining from data stored on the blockchain. In *Business process management workshops,* edited by Chiara Di Francescomarino, Remco Dijkman, and Uwe Zdun, 690–703. Cham: Springer International Publishing. ISBN: 978-3-030-37453-2.

Tran, An Binh, Qinghua Lu, and Ingo Weber. 2018. Lorikeet: a model-driven engineering tool for blockchain-based business process execution and asset management. In *Bpm.*

Weber, Ingo, Christopher Klinkmüller, H. M. N. Dilum Bandara, Richard Hobeck, and Wil M. P. van der Aalst. 2021. Process mining on blockchain data: a case study of augur. In *Business process management,* edited by Artem Polyvyanyy, Moe Thandar Wynn, Amy Van Looy, and Manfred Reichert, 306–323. Cham: Springer International Publishing.

Weber, Ingo, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. 2016. Untrusted business process monitoring and execution using blockchain. In *Business process management,* edited by Marcello La Rosa, Peter Loos, and Oscar Pastor, 329–347. Cham: Springer International Publishing.

Weijters, A. J. M. M., Wil M.P. van der Aalst, and Ana K. A. de Medeiros. 2006. Process mining with the heuristicsminer algorithm.

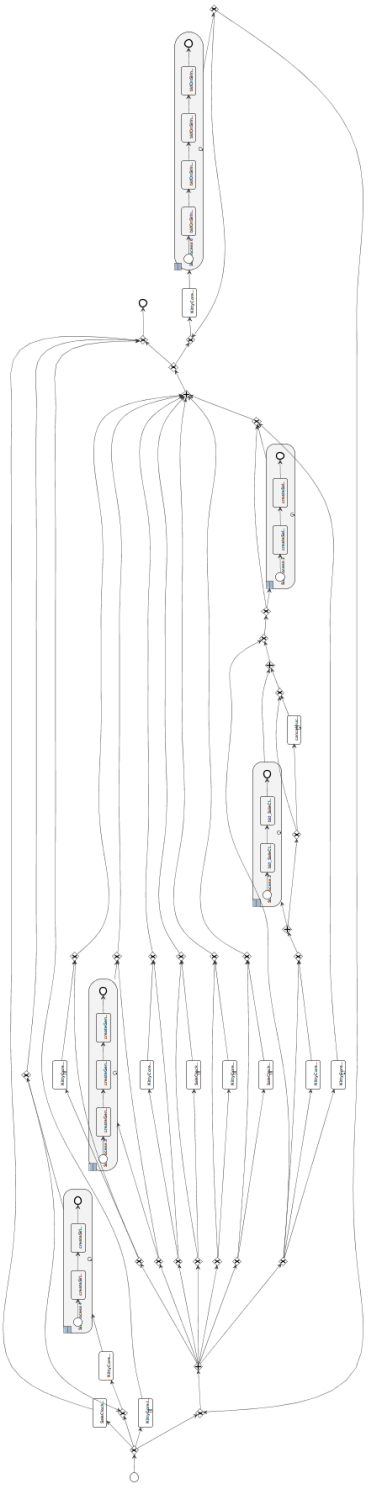

Figure 7. Model discovered with $\text{BPMNMiner}_{\text{Inductive}}$