Future Generation Computer Systems 164 (2025) 107566

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Event log extraction methodology for Ethereum applications

Andrea Morichetta *, Yuri Paoloni, Barbara Re

University of Camerino, Camerino, Italy

ARTICLE INFO

Keywords:
Blockchain
Ethereum
XES
Event log
Internal transactions
Process discovery

ABSTRACT

The adoption of smart contracts in decentralized blockchain-based applications enables reliable and certified audits. These audits allow the extraction of valuable information from blockchains, which can be used to reconstruct the execution of the application and facilitate advanced analyses. One of the most commonly used techniques in this context is process mining, which leverages event logs to trace and accurately represent the process execution of applications. However, extracting execution data from blockchains poses significant challenges, and the current methodologies developed have some limitations. Most approaches are tailored to specific use cases, requiring that analysis techniques are defined during the smart contract's development. Other techniques are applied a posteriori, relying on blockchain events that often lack a standardized format. This absence of standardization requires complex processing steps to correlate logs with the executed actions and such approaches are not universally applicable to all smart contracts on the blockchain, further limiting their scope. Lastly, none of the existing techniques can extract information from event logs embedded in internal transactions of smart contracts.

To address these limitations, we propose EveLog an application-agnostic methodology that can be applied to any EVM-compatible application without predefined constraints. Its primary goal is to extract information from smart contracts, capturing both public and internal transactions, and organizing the results into a structured XES event log. The EveLog methodology consists of five key steps: (i) extraction of data from smart contract transactions, (ii) decoding raw data, (iii) selection of sorting criteria, (iv) construction of traces, and (v) generation of the XES event log. EveLog has been implemented in a client–server application and tested on existing solutions, specifically the CryptoKitties application, a blockchain-based game on the Ethereum blockchain. The study was conducted using 12,996 blocks, including over 8000 real transactions from the Ethereum mainnet.

1. Introduction

Blockchain has recently gained significant attention as an innovative technology for executing business processes and regulating interactions between organizations that do not trust each other. In particular, the use of smart contracts facilitates the execution of complex processes (involving multiple business parties) in a trustless environment, eliminating the need for a third-party authority [1–3]. These advancements have opened new possibilities for business interactions, such as digital asset exchanges, tracking of physical goods, and more. As a result, emerging businesses are increasingly interested in process and data flow analysis techniques to gain insights into how applications are executed. For instance, process mining techniques can generate high-level models from event logs and assess their conformance by comparing the real executions with the expected ones.

Process mining offers significant advantages in the context of blockchain technology [4]. In a domain characterized by decentralized operations and smart contract agreements, understanding process flows and related interactions is crucial for ensuring transparency, efficiency, and trust among participants. With its immutable nature, the blockchain provides a vast amount of data, that can offer valuable insights into how processes are executed when analysed using process mining techniques. This is particularly important in scenarios where multiple organizations interact without a central authority, enabling opportunities for process optimization. Moreover, the certified and immutable nature of blockchain data enhances the reliability of process mining results, making it an ideal tool for organizations aiming for secure, efficient, and transparent operations. However, to fully leverage process mining techniques on blockchain data, it is essential to develop a robust methodology for extracting XES (eXtensible Event Stream) [5] files. This standard ensures that the extracted data adheres to an internationally recognized format, enabling the reuse of existing solutions in the process mining domain.

However, extracting XES logs from blockchains is not straightforward and presents three major challenges [6–8].

* Corresponding author.
E-mail addresses: andrea.morichetta@unicam.it (A. Morichetta), yuri.paoloni@studenti.unicam.it (Y. Paoloni), barbara.re@unicam.it (B. Re).

https://doi.org/10.1016/j.future.2024.107566
Received 10 November 2023; Received in revised form 11 October 2024; Accepted 12 October 2024
Available online 18 October 2024
0167-739X/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

- *Lack of log trace concepts*: Blockchain data does not naturally include the concept of a process identifier to group related records. Indeed, the actions that modify the state of the smart contract are stored as atomic transactions, without native correlation. This characteristic differs from traditional systems (e.g., business process management systems), where an identifier is used to group records associated with the same trace (i.e., belonging to the same process instance).
- *Event data limitations:* Events are not always present in smart contracts and may not be relevant for mining purposes. Extracting meaningful data is particularly complex for applications that were not designed with post-analysis in mind. Often, the available event data is either incomplete or insufficient for significant process mining analysis.
- *Overlooking internal transaction data:* Existing extraction techniques generally ignore internal transactions, focusing exclusively on events generated by smart contracts. However, internal transactions often hold fundamental information needed for a full analysis of the application.

These challenges highlight the need for a specialized methodology to enable effective process mining on blockchain data. Although researchers have proposed various techniques to extract logs from blockchains [4], these methods face limitations in applicability, particularly in environments not originally designed for log extraction. For example, the approach in [9] relies on blockchain-based Business Process Management Systems (BPMS). Here, the access to models implies that the context, the activities, and the execution flow of the process under analysis are known a priori. Such information is typically unavailable for most smart contracts on the blockchain. Other techniques, like [10], focus on collecting blockchain events triggered during the execution of smart contract functions. Similarly, [11] proposes a methodology for generating object-centric event logs. However, blockchain events lack standard parameters and are not always emitted within the body of smart contract functions. A first attempt to group blockchain events into traces using shared parameters (i.e., case ID) was introduced in [12], but without considering information related to internal transactions. To address these challenges, we proposed in this work the following main contributions:

- **EveLog: A new methodology for XES event log extraction.** We introduce a novel application-agnostic methodology for generating XES event logs that considers both public and internal blockchain transactions. This approach is compatible with any Ethereum Virtual Machine (EVM)-based smart contract and requires no specific configurations. As a result, it ensures broad applicability across various blockchain platforms (e.g., Arbitrum, Avalanche, Binance Chain, Polygon, Optimism) that utilize the same transaction format as Ethereum.
- **A Web-based tool implementing the EveLog methodology.** We developed a user-friendly, web-based application that implements the overall methodology. This tool simplifies the extraction and generation of XES event logs from both public and internal blockchain transactions. This tool enhances user access to blockchain data and facilitates its analysis through process mining techniques.

EveLog distinguishes itself by extracting logs from both public[1] and internal blockchain transactions. Public transactions, unlike events, have standardized parameters that provide a common ground for operations. Furthermore, by considering transactions, essential information like *hash*, *sender*, *receiver*, and other transaction fields are always accessible. Internal transactions, on the other hand, represent the underlying execution flow of public transactions, specifically capturing the invocations of smart contract functions involved in the process. Extracting logs from internal transactions creates new opportunities for process analysis, offering significant value for blockchain-based applications. The ability to handle internal transactions is a novel feature not included in existing extraction methodologies. The EveLog methodology consists of five key steps:

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

The EveLog methodology has been implemented in a client–server application and validated using CryptoKitties, a popular blockchain 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 (NFTs) on the Ethereum blockchain. It was chosen for its longevity (deployed in December 2017) and widespread popularity. Additionally, CryptoKitties has also been used to validate other extraction methodologies for event logs in the literature. XES logs generated from CryptoKitties transactions using EveLog were processed through a standard process discovery algorithm. The obtained results were then compared with those from the Ethereum Logging Framework [10], a blockchain extraction framework that relies on blockchain events. Additionally, we evaluated the performance of EveLog in the Cryptokitties case study by conducting a scalability study as an incremental experiment, measuring the time taken as the number of transactions increased. This analysis further confirms the practical applicability of the methodology.

The rest of the paper is organized as follows. Section 2 provides foundational concepts related to blockchain, Ethereum, process mining, and XES standards. Section 3 reviews relevant related works in the literature. Sections 4 and 5 describe the EveLog methodology and its implementation, respectively. Section 6 illustrates the case study of CryptoKitties. Finally, Section 7 wraps up the paper by discussing potential directions for future work.

2. Background

This section presents relevant background information on blockchain technology, with a focus on Ethereum, as well as an overview of process mining, including the XES standard.

2.1. Blockchain

A blockchain is a decentralized ledger technology that records transactions or data securely and tamper-resistantly. It consists of a chain of blocks, each containing a list of transactions or data records. The ledger is replicated across many geographically distributed processing nodes that jointly operate the blockchain system without the central control of any single trusted third party. Ethereum [13] is a decentralized, open-source blockchain with smart contract functionality. Smart contracts are programs stored on a blockchain that run when predetermined conditions are met. They are commonly used to automate agreement processes, ensuring that all participants promptly know the outcome without requiring intermediaries. Similar to traditional programming languages, smart contracts are coded using the Solidity programming language and run on the Ethereum Virtual Machine (EVM). Ethereum is regarded as one of the leading blockchain platforms for smart contracts that can be integrated with a client to develop

[1] The term "public" is used to distinguish standard transactions from "internal" transactions to avoid confusion for the reader.

Decentralized Applications (DApps). Unlike traditional client–server applications, DApps operate on decentralized networks and leverage smart contract functionalities to execute code autonomously. This capability enables DApps to support various decentralized use cases, including governance, gaming, and decentralized finance scenarios.

Listing 1: Example of a Public transaction

```
1 {
2    "LOAD_ID": "2022-05-11 09:20:08.000",
3    "CHAIN_ID": "mainnet",
4    "BLOCK": 6617927,
5    "TIMESTAMP": "2018-10-31 13:59:55.000",
6    "TX_HASH": "0x9af6f20612b239387a051ccb3631826c8c
         ↪ b05458ff5c3fe49b1bd201aea32dbc",
7    "CALL_ID": "\\n",
8    "CALL_TYPE": "call",
9    "FROM_ADDRESS": "0
         ↪ x837ed29de4cab664c550b721bf26dfc028ef6689",
10   "FROM_NAME": "0
         ↪ x837ed29de4cab664c550b721bf26dfc028ef6689",
11   "TO_ADDRESS": "0
         ↪ x06012c8cf97bead5deae237070f9587f8e7a266d",
12   "TO_NAME": "KittyCore",
13   "FUNCTION_SIGNATURE": "0x3d7d3f5a",
14   "FUNCTION_NAME": "createSaleAuction",
15   "VALUE": 0.0,
16   "ARGUMENTS": { "_duration": 172800, "_endingPrice":
         ↪ 390000000000000000, "_kittyId": 949727, "
         ↪ _startingPrice": 450000000000000000 },
17   "RAW_ARGUMENTS": [{ "name": "_kittyId", "raw": "0x00
         ↪ ....000e7ddf", "type": "uint256" }, { "name": "
         ↪ _startingPrice", "raw": "0x00....063
         ↪ eb89da4ed0000", "type": "uint256" }, { "name": "
         ↪ _endingPrice", "raw": "0x0....005698
         ↪ eef06670000", "type": "uint256" }, { "name": "
         ↪ _duration", "raw": "0x00....002a300", "type": "
         ↪ uint256" }],
18   "OUTPUTS": "{}",
19   "RAW_OUTPUTS": "[]",
20   "GAS_USED": 132468,
21   "ERROR": null,
22   "STATUS": true,
23   "ORDER_INDEX": 2630,
24   "DECODING_STATUS": true,
25   "STORAGE_ADDRESS": "0
         ↪ x06012c8cf97bead5deae237070f9587f8e7a266d"
26 }
```

Public transactions are cryptographically signed instructions sent to the blockchain by an account. The simplest example of a transaction is the transfer of cryptocurrencies (e.g., BTC, ETH) from one account to another. A more complex example involves the execution of a smart contract function. Miners process transactions related to smart contracts using the Ethereum Virtual Machine (EVM), which translates the transaction instructions (i.e., Solidity code) into bytecode. Once execution is complete, the transaction is added to a block and propagated throughout the network. As illustrated in Listing 1, a submitted transaction includes several key attributes: *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*, and others.

Internal transactions instead are transactions that occur between smart contracts. This can also include transactions from a smart contract to an external address when sending ETH to a user. These transactions are labelled internal because every deployed smart contract on the Ethereum blockchain has an assigned internal address. Internal transactions are triggered when an external address calls a smart contract to execute an operation. The contract will then use its built-in logic to start interacting with the other required contracts it needs to complete the operation. Even in a single transaction, a smart contract may need to perform several internal calls to other contracts. Unlike public transactions, internal transactions lack a cryptographic signature and are typically stored off-chain, meaning they are not a part of the blockchain. Internal transactions can be retrieved by recording all the value transfers that are part of the execution of an external transaction. The following Listing 2 illustrates an example of the data obtainable for internal transactions using EthTx[2] which is an advanced decoder for blockchain transactions developed by TokenFlow.[3]

Listing 2: Example of an Internal transaction

```
1 {
2    "LOAD_ID": "2022-05-11 09:20:08.000",
3    "CHAIN_ID": "mainnet",
4    "BLOCK": 6617927,
5    "TIMESTAMP": "2018-10-31 13:59:55.000",
6    "TX_HASH": "0x9af6f20612b239387a051ccb3631826c8
         ↪ cb05458ff5c3fe49b1bd201aea32dbc",
7    "CALL_ID": "0_0",
8    "CALL_TYPE": "call",
9    "FROM_ADDRESS": "0
         ↪ xb1690c08e213a35ed9bab7b318de14420fb57d8c",
10   "FROM_NAME": "SaleClockAuction",
11   "TO_ADDRESS": "0
         ↪ x06012c8cf97bead5deae237070f9587f8e7a266d",
12   "TO_NAME": "KittyCore",
13   "FUNCTION_SIGNATURE": "0x23b872dd",
14   "FUNCTION_NAME": "transferFrom",
15   "VALUE": 0.0,
16   "ARGUMENTS": { "_from": "0
         ↪ x837ed29de4cab664c550b721bf26dfc028ef6689",
         ↪  "_to": "0
         ↪ xb1690c08e213a35ed9bab7b318de14420fb57d8c",
         ↪  "_tokenId": 949727 },
17  "RAW_ARGUMENTS": [{ "name": "_from", "raw": "0x000
        ↪ ....0837
        ↪ ed29de4cab664c550b721bf26dfc028ef6689", "type
        ↪ ": "address" }, { "name": "_to", "raw": "0x000
        ↪ ....00
        ↪ b1690c08e213a35ed9bab7b318de14420fb57d8c", "
        ↪ type": "address" }, { "name": "_tokenId", "raw":
        ↪ "0x0000....000000e7ddf", "type": "uint256" }],
18    "OUTPUTS": "{}",
19   "RAW_OUTPUTS": "[]",
20   "GAS_USED": 30507,
21   "ERROR": null,
22   "STATUS": true,
23   "ORDER_INDEX": 2641,
24   "DECODING_STATUS": true,
25   "STORAGE_ADDRESS": "0
         ↪ x06012c8cf97bead5deae237070f9587f8e7a266d"
26 }
```

Events are logs issued during the execution of smart contract functions. They do not have standard parameters and are not mandatory to include in a smart contract function body. Developers can add events at their discretion to log specific actions or outcomes. Typically, events reflect the result of an operation (e.g., transfer of a token, deposit, etc.). The logged data is utilized by external services, like front-end applications, to update their internal states accordingly. The following Listing 3 provides a simple example of Solidity code that demonstrates the use of events to log changes in value (as shown in Line 12).

[2] https://ethtx.info/.

[3] https://tokenflow.live/.

Listing 3: Example of a Solidity event

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

2.2. Process mining

Process mining is a family of techniques related to data science and process management to support the analysis of operational processes based on event logs [14]. In process mining, the process discovery technique maps an event log onto a process model such that the model is representative of the behaviour seen in the event log. Numerous discovery algorithms have been documented in the literature, with the most common being the Heuristic Miner [15], the Inductive Miner [16], and the BPMN Miner [17]. The available process mining algorithms take in input the XES [5] standard. XES defines a grammar for a tag-based language to provide a unified and extensible methodology for capturing event logs. Two widely adopted extensions to XES logs are the *Concept*[4] extension which defines an attribute for storing the generally recognized names of type hierarchy elements across all levels of the XES type hierarchy; and the *Time*[5] extension, which captures the precise timestamp when each event occurs. At the top level of an XES file (as shown in Listing 4), there is one *log* tag (Line 1), which contains all event information related to one specific process. A log can contain an arbitrary number of *trace* tags (Line 2), where each trace represents the execution of an instance of the logged process. Each trace contains *events* (Lines 4, 8, 12, 16), which denote atomic activities observed during the process execution. A *Case ID* is a unique identifier associated with a particular process instance or case being analysed. Each entry in the event log typically corresponds to an activity or event within a process, and the Case ID facilitates the connection of these events to illustrate the execution of a specific process instance or case.

Listing 4: XES example

```
1 <log xes.version="1.0" xmlns="http://code.deckfour.org
      ↪ /xes" xes.creator="Fluxicon Nitro">
2 <trace>
3   <string key="case:concept:name" value="trace 111479"/
      ↪ >
4   <event>
5     <string key="concept:name" value="T1"/>
6     <date key="time:timestamp" value="2011-04-13T14
        ↪ :02:31.199+02:00"/>
7   </event>
8   <event>
9     <string key="concept:name" value="T3"/>
10    <date key="time:timestamp" value="2011-04-13T14
        ↪ :02:31.199+02:00"/>
11  </event>
12  <event>
13    <string key="concept:name" value="T16"/>
14    <date key="time:timestamp" value="2011-04-13T14
        ↪ :02:31.199+02:00"/>
15  </event>
16  <event>
17    <string key="concept:name" value="T4"/>
18    <date key="time:timestamp" value="2011-04-13T14
        ↪ :02:31.199+02:00"/>
19  </event>
20 </trace>
```

3. Related works

Over the years, several attempts for automatically extracting blockchain data for process mining have been proposed [4]. One of the earliest attempts, presented in [9], derives XES logs by analysing the context, activities, and execution flow of the process, but it does not consider internal transactions or their integration. A similar limitation is found in [12], where transactions are extracted, and XES logs are generated by grouping blockchain events into traces based on a shared parameter (i.e., case ID). In contrast, works like [10,18] focus on extracting events emitted by Ethereum smart contracts and formatting them according to XES for process discovery, an approach later extended to include Hyperledger Fabric blockchain support in [19]. Additionally, [11] leveraged transaction traces to capture message calls and contract creations, producing object-centric event logs. These related works reveal a gap in developing a flexible methodology that does not require specialized configuration or deep application-specific knowledge. Furthermore, techniques to capture general information about function executions and their interaction with internal transactions are lacking.

In the following, we focus on the papers most closely related to the EveLog methodology, specifically those that implement log-extraction methodologies for the Ethereum blockchain. Table 1 summarizes the key aspects of such comparison.

In [9], the authors proposed a methodology for extracting logs from transactions involving smart contracts generated by blockchain-based BPMS systems, such as Caterpillar [20] and Lorikeet [21]. These systems convert BPMN collaboration or choreography models into smart contracts, which replicate the execution flow and constraints defined in the models. Caterpillar and Lorikeet employ a factory contract to deploy each generated smart contract, corresponding to a process instance. The factory contract's address is stored in the *"to"* field of the public transaction that creates a new process instance, making it possible to identify the address of each deployed instance. Additionally, key information is known in advance, including (i) the context and operational scenario, (ii) the set of allowed activities, and (iii) the execution flow of the system. Most of the smart contracts deployed on the Ethereum blockchain do not include this information because, even if present, access to the models underlying the smart contract logic would not be possible. As noted, the solution in [9] is applicable only when there is extensive access to the application, which is not always feasible. To overcome this limitation, in EveLog we propose an application-agnostic approach that uses transactions as the basis for log extraction, eliminating the need for detailed access to the analysed application.

The methodology proposed in [9] begins with an activity dictionary containing activity names, function signatures, and function selectors. The function selector is obtained by computing the KECCAK[6] hash (i.e., the cryptographic hash function used by Ethereum)

[4] http://www.xes-standard.org/concept.xesext.
[5] http://www.xes-standard.org/time.xesext.
[6] https://keccak.team/index.html.

Table 1
Main related work overview.

Framework	Extracted data	Scope	Ad-hoc Conf.	App. Knowledge	Disc. Alg. Used
EveLog	Pub. and/or Int. trans.	EVM SC	Not required	Not required	Heuristic M.; BPMN Miner
BPMS [9]	Pub. trans.	Blockchain BPMS	Required	Required	Inductive Visual M.
ELF [10,18]	Blockchain event	Ethereum SC	Required	Required	DFGs, Inductive M.

of the function signatures and saving the hexadecimal of its first four bytes. For instance, the task *"Customer has a problem"* is implemented by a function with signature *"Customer_Has_a_Problem()"*. The selector for this function is *"0xefe73dcb"*, because the KECCAK hash for *"Customer_Has_a_Problem()"* is *"**0xefe73dcb**348c11a7ab31ce1620 102e63c94e84ab393a78f187d1485c8a2c72cc"*. The transaction that indicates an activity enactment is selected using the function selector. Traces are built using the *"to"* field of blockchain transactions. Therefore, every process instance refers to the transactions with a specific smart contract as the recipient. The authors successfully evaluated the methodology on an Incident-management process [22] generating an XES log and applying the Inductive Visual Miner. The methodology has significant limitations in terms of applicability and only applies to the intended target, namely blockchain-based BPMS. It is worth noting that in terms of flexibility, as shown in Table 1, [9] only supports public transactions and requires ad-hoc configuration as well as good application knowledge. On the other hand, with EveLog, we can extract both transaction types and do not require any specific configuration or knowledge.

The framework ELF proposed in [10] is used to extract process event data from applications that utilize the Ethereum's transaction log, specifically Solidity events.[7] The framework comprises three main parts: (i) a *manifest* to specify the target smart contract addresses, the block range, and the transformation rules (i.e., mappings from Ethereum logs to XES data), (ii) an *extractor* that apply the manifest rules, and (iii) a *generator* that generates Solidity code to emit events in line with the manifest specifications. The manifest is required to extract blockchain event data along with the smart contract addresses, block range, and events to be analysed. In addition, a set of rules to transform blockchain events into XES events can be specified in the manifest. 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 do not have standard parameters and are not necessarily present in a function body. Nonetheless, the framework relies on the correct issuance of events exposing the user to incomplete logs and incoherent models. ELF requires smart contracts to extensively emit blockchain events with enough data to create a correlation between them. Each event must contain a shared attribute key to build traces and be emitted in the functions of interest. 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, leading to wrong results (e.g., discovering a model lacking an activity). This aspect 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 organization will redeploy an existing smart contract to make it compliant with the ELF. Deploying smart contracts on Ethereum usually costs thousands of dollars and requires all the services relying on the smart contract to be updated with the new address. As shown in Table 1 also, a deep knowledge of the considered application is needed in this case. Furthermore, the usage of events exposes the problems that have been previously mentioned. In EveLog, the use of transactions ensures that standard parameters are maintained, and they are always involved in the execution of a smart contract function.

4. The proposed evelog methodology

The analysis of existing research highlights a gap in the development of an application-independent methodology for extracting event logs from blockchain transactions. To address this challenge, we introduce EveLog, a novel extraction methodology that efficiently retrieves data from EVM-based smart contracts. This methodology is designed to create XES logs suitable for process analysis techniques without requiring any configurations. EveLog bases its extraction on blockchain transactions, which, differently from using events, gives a common ground for operations. Blockchain transactions have standard parameters that ensure they are always present when collecting data.

The EveLog methodology has been implemented to deal with three different types of transactions.

- **Public transactions** are collected from smart contracts and used to generate XES log according to the selected parameters: sorting criteria, trace identifier (i.e., *case ID*), and concept name.
- **Internal transactions** are collected from smart contracts to generate an XES log, grouping them by the top-level *function name*. Each process log refers to a distinct function, and its traces, which contain internal executions, are grouped by transaction hash.
- **Public and internal transactions** are combined to obtain logs that contain sub-processes. Such logs can be exploited with techniques that can compute processes and sub-processes. In particular, EveLog relies on the *hash*, *from*, *to* and other transaction fields to generate the corresponding XES.

The EveLog methodology, illustrated in the activity diagram in Fig. 1, consists of five steps detailed in the following sub-sections: (i) *data extraction* (Section 4.1), (ii) *raw data decoding* (Section 4.2), (iii) *sorting criteria selection* (Section 4.3), (iv) *trace construction* which considers public (Section 4.4.1), internal (Section 4.4.2), or both type of transactions (Section 4.4.3), and (v) *XES log generation* (Section 4.5).

4.1. Data extraction

In the *data extraction* step, blockchain data is extracted according to two inputs: *smart contract addresses* and *block range*. The first parameter indicates the addresses from which to collect transaction data, while the second parameter is optional and is used to specify the range of blocks to research. More specifically, EveLog requires two addresses for each smart contract: a *transaction address* to collect transactions and an *ABI address* to get the Application Binary Interface (ABI)[8] of the contract. The ABI is essentially the interface of the smart contract, and it is used to encode and decode Solidity smart contract calls for the Ethereum Virtual Machine (EVM) and backward. The ABI is required to decode hexadecimal fields in step 2, which is *"raw data decoding"*. In most cases, the *transaction address* and the *ABI address* coincide, except when a smart contract follows the proxy pattern[9] to allow the upgradability of the smart contract. In such a case, the methodology should collect

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

[8] https://docs.soliditylang.org/en/v0.8.13/abi-spec.html.

[9] https://blog.openzeppelin.com/proxy-patterns/.

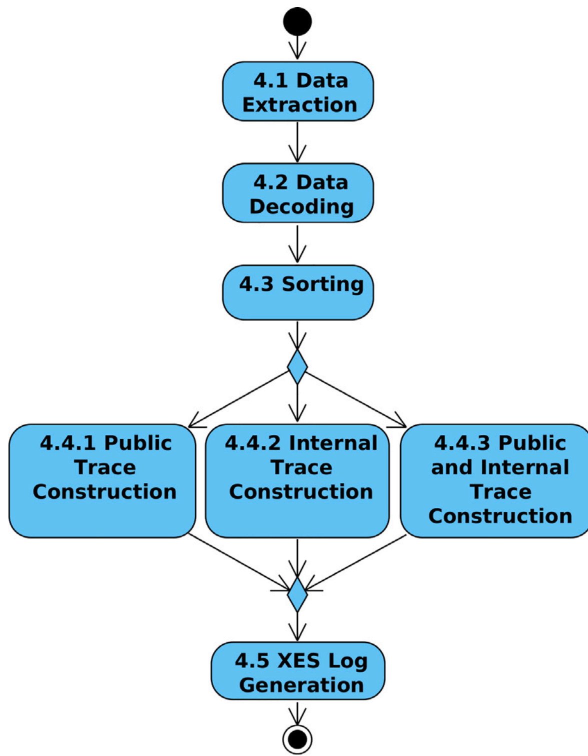


Fig. 1. EveLog Steps.

transactions from the proxy contract and use the ABI of the main contract to decode the hexadecimal fields. EveLog is not limited to one smart contract per execution but supports multiple transaction addresses and ABI address pairs to analyse the transactions of multiple smart contracts together.

To collect **public transactions** we used the blockchain explorer Etherscan.[10] Algorithm 1 shows the pseudocode for extracting public transactions.

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**

Unlike public transactions, **internal transactions** are not stored on the main blockchain. Instead, they occur off-chain and record the values transferred during the execution of public transactions. An advanced blockchain transaction decoder tool such as EthTx[11] is required to collect this data.

EthTx enables the collection of internal transactions, effectively distinguishing them from the public transactions via the *call_id* field. For instance, possible *call_id* values include: *"\\n"*, *"0"*, *"0_0"*, *"0_1"*, *"0_0_0"*, *"0_0_1"*, *"1"*, etc. (see Fig. 2). The *call_id* value for public transactions is "*n*" (line 7, Listing 1), while for internal transactions, it follows a hierarchical pattern. The leftmost number indicates the primary execution, with subsequent numbers denoting nested executions. For instance, *"0_0"* (line 7, Listing 2) occurs within *"0"* but before *"1"*, *"0_0_0"* occurs within *"0_0"* and is executed before *"0_1"*. The algorithm for collecting internal transactions is similar to Algorithm 1, with the only difference of using EthTx instead of Etherscan for data collection.

4.2. Raw data decoding

The raw data collected in the previous step must be decoded before it can be used. For example, some fields in public transactions are in hexadecimal format and require smart contract ABI for their decoding. One of these is the *input* field, which contains the name of the executed function and the respective parameters. The procedure for decoding the *input* is detailed in Algorithm 2.

Firstly, the ABI of the smart contract is acquired (line 1) using the *ABI address* provided during the "data extraction" phase. Subsequently, the algorithm iterates through the public transactions to decode the *input* field. This step decodes and retrieves the name of the function executed in the transaction, along with its parameters, storing them in *function* and *function_params* (Alg. 2 lines 2–6). Additionally, any fields from the transaction that are not required for the subsequent steps, such as *nonce*, *value*, *isError*, are removed (line 7). This decoding process ensures that relevant information from the *input* field is extracted and made readily available for further analysis.

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: $transactions.remove([\text{"}nonce\text{"}, \text{"}value\text{"}, \text{"}isError\text{"}...])$

The decoding step is not performed on internal transactions since the data obtained with *EthTx* contains the raw and already decoded fields. Also in this case, unnecessary fields are removed: *load_id*, *chain_id*, *raw_arguments*, *raw_outputs*, *error*, etc.

4.3. Sorting criteria selection

The sorting step is the same for public and internal transactions. In this phase, fields are selected from the available set of transaction fields. The selection criteria include not only the standard fields left over from the cleaning phase of the raw data but also the previously extracted function parameters. However, it is worth noting that transactions in the same block share the same timestamp, reflecting the time the block is validated rather than when the transaction was sent. However, public transactions have the *transactionIndex* field, which indicates the order in which they are executed in a block.[12] By default, EveLog sorts transactions by combining both *timestamp* and *transactionIndex*. However, it also supports alternative criteria that use one or more of the available transaction fields.

4.4. Trace construction

The *trace construction* is a crucial step that allows the building of the XES log and influences the process analysis technique that will be applied to it. In particular, the selection of the field representing the *case ID* strongly influences the output XES log and allows the highlighting of different aspects. In the following subsections, we will discuss how the trace construction algorithm adapts based on the three data inputs supported by the EveLog methodology.

[10] https://etherscan.io/.
[11] https://ethtx.info/.
[12] https://ethereum.org/en/developers/docs/apis/json-rpc/.

The contract call From 0x95222290...5CC4BAfe5 To 0x859ee23A...45d5E4806 produced 2 Internal Transactions — ADVANCED MODE:

Type Trace Address	From		To	Value	Gas Limit
staticcall_0_1	0x859ee23A...45d5E4806	→	0x3c55986C...cF9B7c03F	0 ETH	5,169
delegatecall_0_1	0x859ee23A...45d5E4806	→	0x191A8f7f...d848155c7	0.037860496447140158 ETH	92

Fig. 2. Example of internal transactions.

Raw Data Transaction		XES Log
{"from": "0x837…6689"}	→	<string key="case:concept:name" value="0x837…6689" />
{"timestamp": "2018-10-31 13:59:55.000"}	→	<date key="time:timestamp" value="2018-10-31T13:59:55" />
{"function_name": "createSaleAuction"}	→	<string key="concept:name" value="createSaleAuction" />

Fig. 3. Mappings from raw data to XES selecting *"from"* as *"case ID"*; *"timestamp"* as *"time:timestamp"* and *"function_name"* as *"concept:name"*.

4.4.1. Public transactions

An identifier (i.e., *case ID*) must be selected from the list of available transaction fields for the trace construction phase of public transactions. As explained in Section 1, blockchains do not provide a built-in trace identifier, so the user must select the most appropriate one based on the desired outcome. For instance, if the field *"from"* (i.e., the sender of the transaction) is selected as *case ID*, then the public transactions sent by that specific address will be included in a trace. Process discovery with such a log creates a model representing the lifecycle of public transactions sent from 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: $public_transactions.concept_name = public_transactions.inputFunctionName$

Algorithm 3 shows a possible manipulation of records to generate traces. In addition to the *case ID*, other fields related to the *Concept*[13] and *Time* XES extensions are set:

- **case:concept:name**: groups events in traces (see line 3, Listing 4). It is the equivalent of the *case ID* in XES logs.
- **concept:name**: gives the name to XES events (see lines 5,9,13,17, Listing 4). For instance, in models generated with process discovery, it is the name of the activities.
- **time:timestamp**: used in XES logs and process mining techniques as the standard timestamp field (see lines 6,10,14,18, Listing 4).

In Alg. 3, the *"from"* field is mapped to *case:concept:name* (i.e., *case ID*) to generate traces containing all 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 after the smart contract function executed in the public transaction.

Fig. 3 provides a graphical representation of the mapping from raw data transactions to XES log. To get a better overview of the resulting output, readers can take a look at the example in Listing 1.

4.4.2. Internal transactions

The trace construction for internal transactions is a bit more complex, and a dedicated algorithm is needed (see Algorithm 4). As mentioned at the beginning of this section, the objective is to create a separate log for each *function name* identified in the input data. The *function name* refers to the name of the smart contract function executed in a public transaction. Each generated XES log will contain traces of internal transactions that refer to a specific function (i.e., *function name*) performed within a public transaction.

The data obtained with EthTx contains both public and internal transactions. Each record has a *tx_hash* (line 6, Listing 2) field that indicates the public transaction to which it belongs. This field is especially useful in understanding to which public transaction an internal one belongs. In each created log, the traces will represent the internal executions of the public transactions; therefore, internal transactions should be grouped in traces using the public transaction hash. The *call_id* (line 7, Listing 2) field can be used to create different logs for each *function name* (line 14, Listing 2) and distinguish public and internal transactions.

The algorithm 4 iterates through the list of public and internal transactions collected with EthTx (lines 4–12) and stores 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. It is ideal for the *"group by"* operation executed in line 13. In the same iteration, the algorithm stores the value of the variable *user_address* (i.e., sender address) in the field *origin_address* for each record (line 11). This is done because, on internal transactions, the *from_address* field refers to the previous smart contract in the execution chain and not to the sender of the public transaction. Moreover, it helps 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., *transaction.from_name*) (line 9). The prefix helps identify the smart contract that performs a function, as it can happen that function names are similar among smart contracts of the same application. The last step is to split the records into different sets, each containing the internal transactions for a specific *function name* (lines 13–16). This is achieved by performing a *group by tx_hash* and mapping *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). To get a better overview of the resulting XES output, readers can take a look at the example in Listing 6.

4.4.3. Combination of public and internal transactions

The Algorithm 5 presets the procedure for combining public and internal transactions. The objective of this phase is to create a standardized format for XES logs that can be effectively processed by process mining algorithms designed to handle sub-processes. The resulting logs

[13] http://www.xes-standard.org/concept.xesext.

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**

meet the requirements of a specific algorithm (BPMN Miner [17]), as demonstrated later in Section 6. The algorithm 5 takes in input the dataset collected with EthTx, which contains both public and internal transactions. The strategy involves adding a new field to the record of internal transactions with a key equal to the name of the function executed in the public transaction and a value equal to the hash of the public transaction (line 13) (e.g., *"transfer": "0x..."*). The new field creates a relationship between internal transactions belonging to the same public transaction. Furthermore, the *function_name* of the records with *call_id* = *"\\n"* (i.e., public transactions) is prefixed with the smart contract name (i.e., *to_name*) (line 9), while the *function_name* of records with *call_id != "\\n"* (i.e., internal transactions) 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 key *id* is added to both public and internal transactions to group them in traces (line 15). The value of *id* changes based on the selection criteria for generating 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 their corresponding internal transactions. Finally, there are two groupings in the data: (i) the *id* field used to group records in traces, and (ii) the new field (i.e., the assignment done in line 13) to identify sub-processes. To get a better overview of the resulting XES output, readers can take a look at the example in Listing 8.

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**

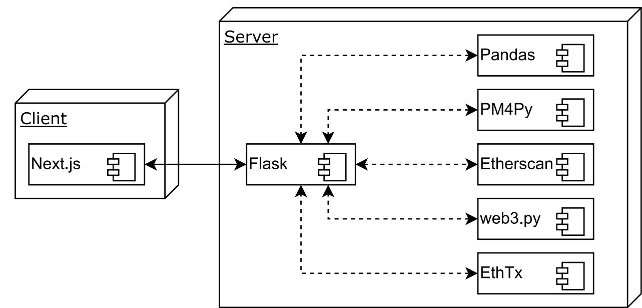


Fig. 4. EveLog architecture.

4.5. XES log generation

After the trace construction step, the EveLog methodology is finally ready to generate the XES log based on the parameters selected by the user in the previous steps. The generated XES logs use two additional extensions:

- The *Concept*[14] extension defines an attribute that stores the generally understood name of type hierarchy elements for all levels of the XES type hierarchy.
- The *Time*[15] extension defines an attribute that stores the precise timestamp at which the event occurred.

5. EveLog Implementation

In this section, we focus on the technological and architectural decisions that shaped the implementation of EveLog. The EveLog implementation for collecting public transactions has been implemented using Python and integrated into a client–server application. The tool is available at https://bit.ly/478xSAU. The steps for internal transactions were not included, as they require a full Ethereum node with the debug option, to which public access cannot be granted. The application allows the collection of public transactions and generates XES logs for one or more smart contracts and a specified block range.

Fig. 4 illustrates the client–server application's structure. The *Next.js*[16] framework is used to render content and make requests to the server in the client. The server handles requests using *Flask*,[17] which then routes them to the appropriate service, e.g., *Pandas*[18] for processing data, *Etherscan* for fetching transactions, and *PM4Py*[19] for building traces and generating XES logs.

The application has three pages: the *home page*, the *about page*, and the *tool page*. The *about page* provides information on filling out forms, generating data, and more. The *tool page* includes two steps: transaction collection and XES log generation. In the first step, shown in Fig. 5, users specify information about the smart contracts from which to collect transactions and the block range. The considered information is reported in the following.

- *Name*: the name of the smart contract. Although it is not mandatory to use the actual name of the smart contract specified in the code, the name helps identify the smart contract to which the collected transactions and generated logs belong.

[14] http://www.xes-standard.org/concept.xesext.
[15] http://www.xes-standard.org/time.xesext.
[16] https://nextjs.org/.
[17] https://flask.palletsprojects.com/en/2.2.x/.
[18] https://pandas.pydata.org/.
[19] https://pm4py.fit.fraunhofer.de/.

Home Tool About

Insert the smart contracts you want to collect data from

For more information on how to use the tool check the About page

Name: SaleClockAuction | Transactions address: 0xb1690c08e213a35ed9bab7b318de14420fb57d8c | ABI address: 0xb1690c08e213a35ed9bab7b318de14420fb57d8c

Name: KittyCore | Transactions address: 0x06012c8cf97bead5deae237070f9587f8e7a266d | ABI address: 0x06012c8cf97bead5deae237070f9587f8e7a266d

Start block: 6605101 | End block: 6618097

SEARCH

Preview results for SaleClockAuction, KittyCore


```
[
  {
    "SaleClockAuction__tokenId": "1135837",
    "blockHash": "0xed39b175bc99a39280658c697e1ec2c9e431be6327148ab3c10dededca8efe18",
    "blockNumber": "6605101",
    "confirmations": "10100660",
    "contractAddress": "",
    "cumulativeGasUsed": "4497226",
    "from": "0x8949db9fbb4716ce5a2803085c7732c14fe03a37",
    "functionName": "bid(uint256 _tokenId)",
    "gas": "152016",
    "gasPrice": "8800000000",
    "gasUsed": "56344",
    "hash": "0x72288ba1555b9a4773a1ad3594958da4f25b7215abc71fa0cb33def06d3a04ee",
    "input": "0x454a2ab30000000000000000000000000000000000000000000000000000000000001154dd",
    "inputFunctionName": "bid",
    "inputFunctionParams": "{'_tokenId': 1135837}",
    "isError": "0",
    "methodId": "0x454a2ab3",
    "nonce": "535",
    "timeStamp": "1540813622",
    "to": "0xb1690c08e213a35ed9bab7b318de14420fb57d8c",
    "transactionIndex": "118",
```


Download full dataset | Next step

Fig. 5. EveLog transactions collection.

- *Transaction address*: the contract address from which transactions will be collected.
- *ABI address*: the smart contract address that 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), shown in Fig. 6, uses the transactions collected in the previous step and generates the final XES according to the following parameters.

- *Sort by*: the parameter by which records are sorted.
- *Case ID*: the parameter selected to build traces. For instance, if *"from"* is selected as the Case ID, the resulting log will contain a separate trace for each unique user interacting with the smart contract.
- *Concept name*: the *case:concept:name* value in the XES event is used in process discovery to represent the name of the activities included in the output model.

The user can choose sorting parameters through a dropdown menu, which includes both standard Ethereum transaction fields and parameters derived from the smart contract functions executed within the transactions. Once the process is completed, the resulting XES file can be downloaded, enabling users to utilize the log for further process mining analysis.

6. Case study: Cryptokitties

In this section, we present a case study on CryptoKitties, a popular Ethereum-based game where users can buy, breed, and trade virtual cats. Using EveLog, we extracted XES logs from CryptoKitties transactions and analysed these logs using process discovery algorithms. The results were evaluated against the ELF framework [10], which, as discussed in Section 3, focuses on generating XES logs by extracting blockchain events. This comparison highlights the effectiveness and advantages of EveLog in capturing both transactions and internal events in a more comprehensive way.

Listing 5: XES trace generated from public transactions

```
1 <trace>
2   <string key="case:concept:name" value="0
        x8949db9fbb4716ce5a2803085c7732c14fe03a37"/
        >
3   <event>
4     <int key="blockNumber" value="6605106"/>
5     <string key="hash" value="0x233fa05e1465...."/>
```

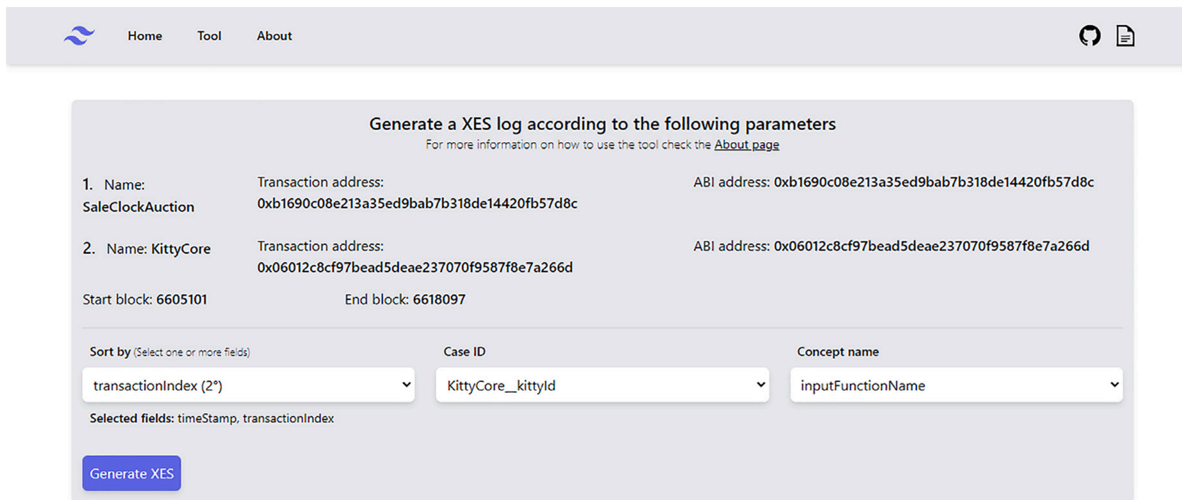


Fig. 6. EveLog XES log generation.

```
6     <string key="blockHash" value="0x03e8bcb...."/>
7     <int key="transactionIndex" value="68"/>
8     <string key="from" value="0x8949db9fbb...."/>
9     <string key="to" value="0x06012c8cf...."/>
10    <int key="value" value="0" />
11    <string key="input" value="0x3d7d3f5a0...000278d00"
          ↪ />
12    <string key="methodId" value="0x3d7d3f5a" />
13    <string key="functionName" value="
          ↪ createSaleAuction(uint256 _kittyId, uint256
          ↪ _startingPrice, uint256 _endingPrice,
          ↪ uint256 _duration)" />
14    <string key="inputFunction" value="Function
          ↪ createSaleAuction(uint256,uint256,uint256,
          ↪ uint256)" /> <string key="inputFunctionName"
          ↪ value="createSaleAuction"/>
15    <string key="inputFunctionParams" value="{'
          ↪ _kittyId': 1135862, '_startingPrice':
          ↪ 197000000000000000, '_endingPrice':
          ↪ 197000000000000000, '_duration': 2592000}"/
          ↪ >
16    <date key="TIMESTAMP" value="2018-10-29T11:48:27" /
          ↪ > <string key="org:resource" value="0
          ↪ x8949db9fbb4716ce5a2803085c7732c14fe03a37"
          ↪ />
17    <date key="time:timestamp" value="2018-10-29T11
          ↪ :48:26" />
18    <string key="concept:name" value="
          ↪ createSaleAuction" />
19   </event>
20   ...
21 </trace>
```

CryptoKitties is a decentralized application (DApp) built on the Ethereum blockchain, enabling users to manage virtual cats known as CryptoKitties. These CryptoKitties are unique digital assets, each defined by a distinctive set of attributes, or "genes", which shape their appearance and characteristics. Users can breed their CryptoKitties to create new cats, combining attributes from both parents. In addition to breeding, users can buy and sell CryptoKitties using Ether (ETH), Ethereum's cryptocurrency. The value of a CryptoKitty is determined by its traits and rarity, with some fetching prices in the hundreds of thousands of dollars. As one of the earliest and most successful examples of NFTs (non-fungible tokens) on Ethereum. CryptoKitties was

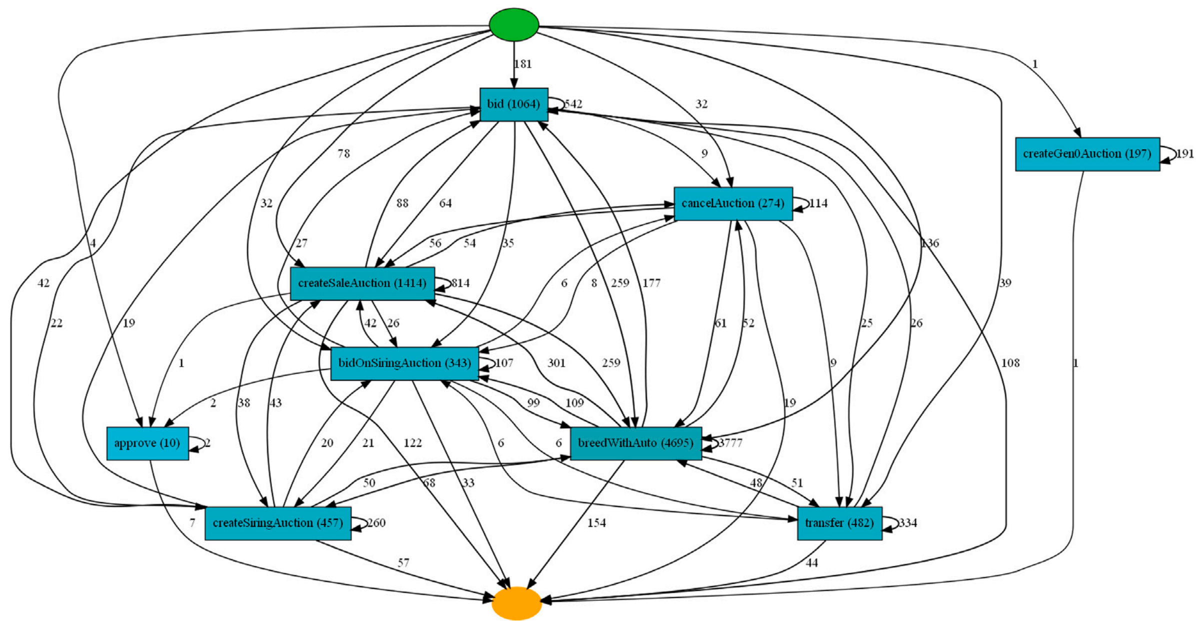


Fig. 7. Heuristic Miner on CryptoKitties log generated with EveLog.

chosen for this case study due to its longevity (deployed in December 2017), widespread popularity, and its frequent mention in the literature regarding event log extraction methodologies [10].

The EveLog was executed on CryptoKitties using the *SaleClockAuction*[20] and *KittyCore*[21] smart contracts and a block range from *6605101* to *6618097*. These configurations correspond to those employed in [10] and were selected to ensure consistency during the evaluation process. The process of *data extraction*, *raw data cleaning,* and *sorting* is similar for public transactions, internal transactions, and their combination. Instead, the *trace construction* algorithm was adapted based on the specific input data. After extracting and processing the data, the Heuristic Miner [15] was applied to the resulting logs with default PM4Py parameters. The following thresholds were used: *"dependency threshold"*: 0.5, *"and threshold"*: 0.65, *"loop two threshold"*: 0.5. In terms of performance, EveLog took 23.26 s to complete data extraction for both smart contracts and 58.99 s to generate the traces. For further details regarding performance and evaluation metrics, refer to Section 6.4.

6.1. Public transactions

This section presents the evaluation of EveLog against ELF. As detailed in Section 3, ELF is a log-extraction framework designed to map blockchain events to XES logs, requiring a manifest file containing smart contract addresses, block ranges, and mappings of blockchain events to XES events. Using the manifest, ELF can selectively extract certain blockchain events and map each to multiple XES events. In both EveLog and ELF executions, the process instance is the *id* of a cryptoKitty. The extraction process with EveLog, using the configuration mentioned earlier, generated a log composed of *545* traces and *8936* events. An example trace from the extracted log is provided in Listing 5.

Figs. 7 and 8 illustrate the process models obtained from logs generated by EveLog and ELF, respectively. The model in Fig. 7 is an Heuristic net discovered using the PM4Py's Heuristic Miner, while the model in Fig. 8 is a Directly-Follows Graphs generated using ProM taken from [10]. Both models offer a clear and intuitive representation of activity flows within a process, making them valuable tools for analysing and exploring process data. They are particularly useful for identifying opportunities for process improvement and optimization. In these visualizations, squared nodes represent process activities, and directed edges show the sequence of activity execution. An edge from activity A to activity B indicates that B directly follows A in the log, with the edge weights representing the frequency of such transitions. As mentioned earlier, the cryptoKitty ID serves as the trace identifier, allowing both models to depict the lifecycle of transactions associated with a cryptoKitty within the selected block range.

The two models are quite similar, with the main difference being the names of the activities in the generated logs.

- EveLog (9 activities): *bid, createGen0Auction, cancelAuction, createSaleAuction, bidOnSiringAuction, breedWithAuto, approve, transfer, createSiringAuction*.
- ELF (9 activities): *AuctionCreated, AuctionSuccessful, AuctionCancelled, Birth, Pregnant, KittybecameMother, Transfer, SireMadeSomeonePregnant, KittyBecameFather*.

As mentioned, ELF [10] supports mappings from a blockchain event to multiple XES events. Furthermore, certain events can be omitted, as happened with the event *Approval*[22] emitted in the *approved* function. The aforementioned justifies the difference in the activities between the two methodologies. The naming difference can be justified because events and functions can have different names in Solidity. 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, on Ethereum, events are bound to functions. A function can be in the Solidity code without an event, while events can be present only within a function body. The same event can be emitted in different functions, possibly with different meanings, and could lead to incorrect results if not mapped to multiple XES events. An example is *Pregnant* which, in [10], is mapped to *SireMadeSomeonePregnant* to comprehend the *bidOnSiringAuction* function. The EveLog methodology automatically catches two XES events, while ELF needed the mapping from *Pregnant* to *SireMadeSomeonePregnant*.

[20] https://bit.ly/45OD9MH.
[21] https://bit.ly/3QJ61lv.
[22] https://bit.ly/3tR6R6J.

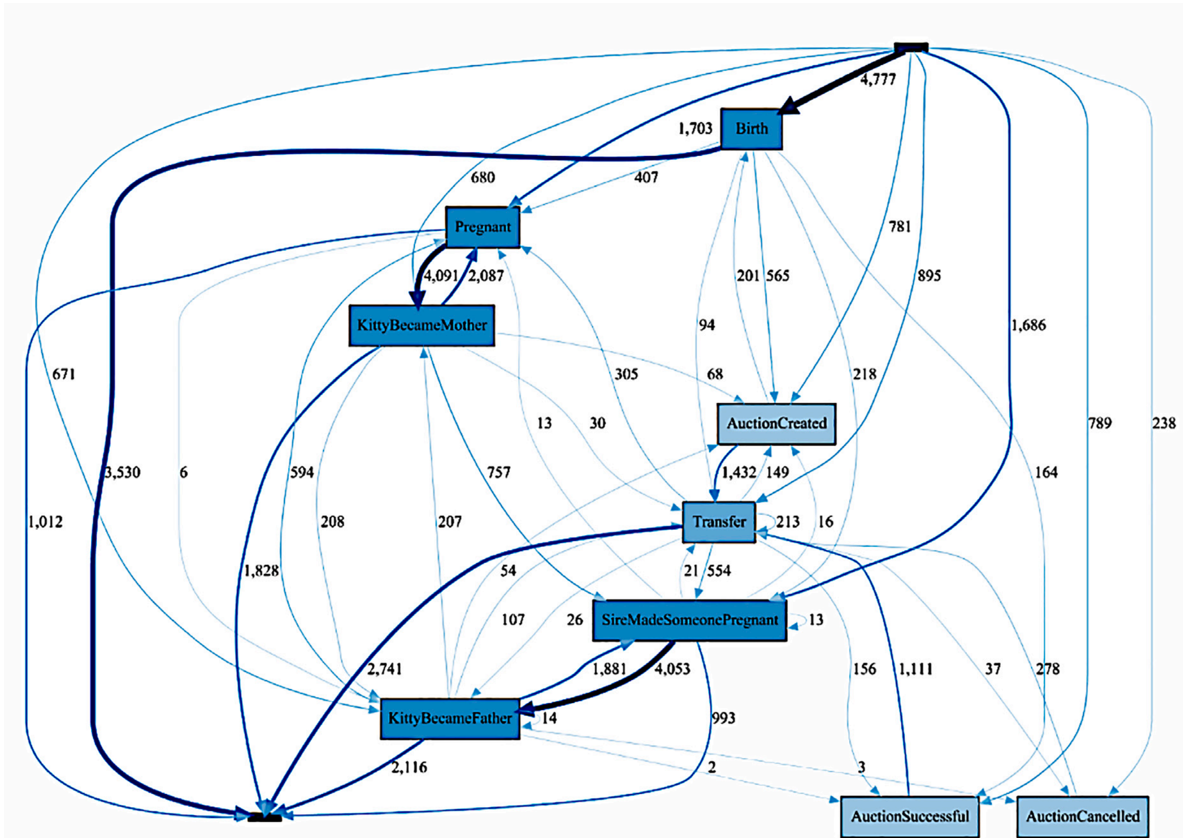


Fig. 8. ProM's Directly-Follows Graph discovered on the ELF-generated log.
Source: The model was taken from [10].

The ELF approach requires extensive configurations and the creation of a manifest, which involves a detailed understanding of the smart contract's code. This makes the process effort-prone and error-prone due to the need for manual setup and the risk of incorrect mappings. In contrast, EveLog significantly simplifies the extraction process, requiring only the smart contract's address and the block range to function. This reduces the complexity and eliminates the need for prior knowledge of the smart contract's internal logic. The evaluation conducted was not intended to provide a direct comparison with ELF, but rather to confirm the flexibility and consistency of EveLog. It has shown that comparable results can be obtained with far fewer configurations and without needing prior insight into the input smart contracts. Additionally, it can be applied to any DApps, even those that do not emit blockchain events, offering broader applicability.

6.2. Internal transactions

As described in Section 4.4.2, the *trace construction* step for internal transactions is designed to generate a separate log for each function name identified within the dataset. In this specific case, we generated *four* logs based on the previously mentioned configurations. Each log captures the internal transactions associated with public transactions that execute a specific function, allowing for a granular view of the process execution at the function level. Such public functions were *bidOnSiringAuction*, *createGen0Auction*, *createSaleAuction*, and *createSiringAuction*. The *hash* of the public transaction was selected as *case ID*, and the *function name* as *concept:name* for each trace. Thus, each trace represents the internal execution flow of a public transaction. An example trace is shown in Listing 6. 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 (line 2). The events have the *concept:name* attribute (line 19) equal to the *function_name* attribute (line 13).

When the Heuristic Miner is applied to these logs, the resulting models provide insights into how the smart contract functions are executed. The splits in the model correspond to decision constructs (e.g., *if*, *else*) or loops (e.g. *for* or *while*). If the model reveals rare variants or unusual execution paths, these may indicate potential bugs or inefficiencies in the smart contract code. Identifying such anomalies can be valuable for debugging and optimizing the smart contract's functionality.

Listing 6: XES trace generated from internal transactions

```
1 <trace>
2   <string key="case:concept:name" value="0
        ↪ x233fa05e1465b675e...."/>
3     <event>
4       <int key="block" value="6605106"/>
5       <string key="tx_hash" value="0x233fa05e1465b675e
            ↪ ...."/>
6       <string key="call_id" value="0"/>
7       <string key="call_type" value="call"/>
8       <string key="from_address" value="0x06012c8cf97be
            ↪ ...."/>
9       <string key="from_name" value="KittyCore"/>
10      <string key="to_address" value="0
            ↪ xb1690c08e213a35e...."/>
11      <string key="to_name" value="SaleClockAuction"/>
12      <string key="function_signature" value="0
            ↪ x27ebe40a"/>
13      <string key="function_name" value="createAuction"
            ↪ />
```

```
14      <string key="arguments" value='{"_duration"
          ↪ :2592000, "_endingPrice":197000," _seller"
          ↪ : "0
          ↪ x8949db9fbb4716ce5a2803085c7732c14fe03a37
          ↪ ", "_startingPrice":197000," _tokenId"
          ↪ :1135862}' />
15      <string key="outputs" value="{}" />
16      <int key="order_index" value="2088" />
17      <string key="origin_address" value="0
          ↪ x8949db9fbb4716ce5a28...." />
18      <date key="time:timestamp" value="2018-10-29T11
          ↪ :48:26+00:00" />
19      <string key="concept:name" value="createAuction"
          ↪ />
20    </event>
21    ...
22 </trace>
```


The model for the ‘bidOnSiringAuction‘ function is presented in Fig. 9 and the corresponding Solidity code is shown in Listing 7. In the model, the first internal transaction is the *getCurrentPrice*. Afterwards, the model splits into two paths: (i) the execution terminates directly, or (ii) continues with the *bid* function. This behaviour is also evident in the code (line 11) through a *require* control statement, which can stop the function’s execution. Going ahead, the functions *bid* (line 12) and *_breedWith* (line 13) 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*. These functions’ implementation is available on Etherscan.[23]

Listing 7: bidOnSiringAuction function Solidity implementation

```
1  function bidOnSiringAuction(uint256 _sireId,
2    uint256 _matronId)
3    external
4    payable
5    whenNotPaused
6    {
7      require(_owns(msg.sender, _matronId));
8      require(isReadyToBreed(_matronId));
9      require(_canBreedWithViaAuction(_matronId,
          _sireId));
10     uint256 currentPrice = siringAuction.
          getCurrentPrice(_sireId);
11     require(msg.value >= currentPrice+autoBirthFee);
12     siringAuction.bid.value(msg.value - autoBirthFee)
          (_sireId);
13     _breedWith(uint32(_matronId), uint32(_sireId));
14   }
```

6.3. Combination of public and internal transactions: trace construction

As described in Section 4.4.3, combining public and internal transactions requires a process mining algorithm capable of handling processes and sub-processes. For this purpose, BPMN Miner [17] was selected to take advantage of this combination. BPMN Miner is an algorithm that can generate a BPMN model featuring sub-processes, boundary events, and activity markers. The underlying mechanism of BPMN Miner is similar to the concepts of primary and foreign keys in relational databases. It splits the log into sub-logs based on process instance identifiers (i.e., keys) and references from sub-processes to parent process instances (i.e., foreign keys). This approach allows the

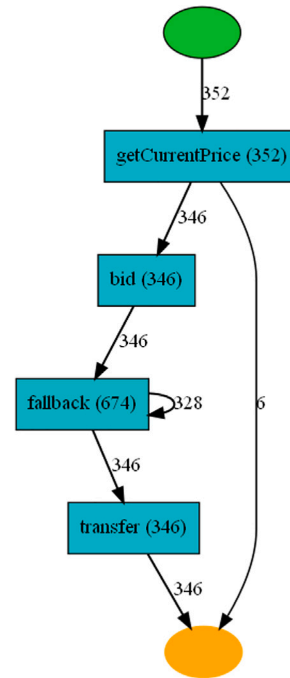


Fig. 9. Model discovered from the internal transactions of *bidOnSiringAuction*.

clear representation of both main processes and their sub-processes within the same model. For a more detailed explanation of BPMN Miner, please refer to [17].

Some changes need to be made during the log generation to apply BPMN Miner to the extracted log. The algorithm allows the selection of the keys for which sub-processes are referenced. The *function_name* of 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 based on the artificial log contained in the BPMN Miner source code.[24] In particular, the *from* field was selected as *case ID*, and the *function name* as *concept:name*. As described in Section 4.4.3, 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 presented in Listing 8.

The *case:concept:name* (line 2) of the trace (i.e., *case ID*) is equal to the *id* attribute in the events belonging to the trace (line 4). 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 also prefixing the *function name* of the parent transaction (e.g., *bidOnSiringAuction_KittyCore.getCurrentPrice*) to mark the parent process to which the event belongs (line 12). The events representing internal transactions contain an attribute with _ as the prefix to indicate that they are internal transactions during the BPMN Miner execution. The *trace construction* step generated a log with *476* traces, *13946* events, and *4* different event types. Fig. 10 illustrates the model discovered with the $\text{BPMNMiner}_{\text{Inductive}}$ algorithm. The resulting model is a BPMN diagram that integrates both public and internal transactions. Standalone activities situated outside of sub-processes represent public transactions, while activities that are enclosed within sub-processes correspond to internal transactions. Each sub-process is explicitly linked to a specific public transaction, which is indicated in the sub-process name.

[23] https://bit.ly/3Shotm5.

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

Listing 8: XES trace generated for BPMN Miner

```
1 <trace>
2   <string key="case:concept:name" value="0x8949db9f
      ↪ ...."/>
3     <event>
4       <string key="id" value="0x8949db9f...."/>
5       <date key="time:timestamp" value="2018-10-29T11
          ↪ :48:26+00:00"/>
6       <string key="concept:name" value="KittyCore.
          ↪ createSaleAuction"/>
7     </event>
8     <event>
9       <string key="id" value="0x8949db9f...."/>
10      <string key="createSaleAuction" value="0
          ↪ x233fa05e1465b675e...."/>
11      <date key="time:timestamp" value="2018-10-29T11
          ↪ :48:26+00:00"/>
12      <string key="concept:name" value="
          ↪ createSaleAuction_KittyCore.
          ↪ createAuction"/>
13    </event>
14    <event>
15      <string key="id" value="0x8949db9f...."/>
16      <string key="createSaleAuction" value="0
          ↪ x233fa05e1465b675efc20abab3f9331...."/>
17      <date key="time:timestamp" value="2018-10-29T11
          ↪ :48:26+00:00"/>
18      <string key="concept:name" value="
          ↪ createSaleAuction_SaleClockAuction.
          ↪ transferFrom"/>
19    </event>
20    <event>
21      <string key="id" value="0x8949db9f...."/>
22      <date key="time:timestamp" value="2018-10-29T11
          ↪ :54:38+00:00"/>
23      <string key="concept:name" value="KittyCore.
          ↪ createSaleAuction"/>
24    </event>
25 ...
26 </trace>
```

6.4. Performance evaluation

This section evaluates the performance of EveLog on the CryptoKitties smart contract through an incremental analysis based on the number of transactions considered. The EveLog methodology is hosted on a virtual machine with 4 cores (E5-2620@2.40 GHz) and 4Gb of RAM. Table 2 presents the performance metrics for extracting and generating traces from the *CryptoKitties: Sales Auction* smart contract.[25] We measured performance by progressively increasing the number of transactions and assessing two primary metrics for each experimental setup: *average extraction time (s)* and *average generation time (s)*. The results reported are based on the average of ten iterations of the same experiment. The Number of Transactions (N° of trans.) column specifies the total number of transactions processed in each request. It ranges from a single transaction to over one million transactions, with the latter being the maximum recorded for the CryptoKitties smart contract. The *Avg. Extraction T. (s)* represents the duration required to retrieve data from the Ethereum mainnet, which increases linearly with the transaction count. Starting with 2.97 s for a single transaction, it reaches approximately 596.12 s for 1,196,109 transactions. This trend highlights the growing complexity as the number of transactions increases. In contrast, *Avg. Generation time (s)*, measures how long EveLog takes to produce the XES output from the extracted data. For instance, processing one transaction takes 0.42 s, while processing 1,196,109 transactions increases the time to 2160 s. This trend indicates that trace generation is computationally intensive, particularly as the dataset size grows. Specifically, the generation time is also influenced by the selected filters. In our experiments, we sorted transactions by their *timestamps* and *transactionIndex*, we used the *_tokenId* as Case ID and *inputFunctionName* as the Concept name. This analysis provides insights into the scalability and performance capabilities of EveLog when handling large datasets. The results are promising and demonstrate that the framework remains robust and effective, even with large volumes of data.

[25] *0xb1690C08E213a35Ed9bAb7B318DE14420FB57d8C*.

Table 2
EveLog performance evaluation.

N° of trans.	Avg. Extr. T. (s)	Avg. Gen. T. (s)
1	2.97	0.42
10	3.28	0.63
50	3.45	1.11
100	3.65	1.42
500	4.25	4.9
1,000	5.13	5.42
10,000	14.23	41.25
50,000	114.00	189.23
100,000	184.15	413.00
1,196,109	596.12	2,160.00

7. Conclusion and future works

This paper introduces a novel extraction methodology, EveLog, designed to facilitate the extraction of XES logs from blockchain data. The key feature of EveLog is its application-agnostic nature, allowing it to be used broadly across various smart contracts without prior configurations or adjustments. The methodology focuses on blockchain transactions, which, in contrast to events, possess standard parameters consistent across different smart contracts, ensuring reliability and uniformity in the extraction process. EveLog is designed to be highly adaptable and can be applied to various EVM-compatible blockchains, as they share the same transaction format as Ethereum [23]. A custom adapter is required to handle their transaction payloads for non-EVM blockchains (e.g., Algorand, Bitcoin, Cardano, Solana). However, the core methodology of EveLog remains consistent across different blockchain environments. Extracting XES logs from internal transactions is challenging because they are not stored on-chain, and only a limited number of tools support their collection. However, extracting logs from internal transactions enables the inspection of execution traces in immutable applications, such as smart contracts. This capability can significantly aid in discovering bugs and preventing exploits, which is particularly valuable in blockchain contexts where assets like cryptocurrencies and decentralized finance are managed, involving considerable value and risk. The case study on CryptoKitties demonstrates the applicability of the proposed methodology in a real-world application. Process discovery was successfully applied to the generated logs, highlighting the effectiveness of the methodology. However, the list of applicable techniques extends beyond process discovery. Any technique supporting XES as input can be applied to the generated logs, including conformance checking, social network analysis, user/usage profiling, simulations, and even more advanced applications such as creating digital twins, as suggested in [24].

The proposed methodology paves the way for addressing the challenges of applying process mining to blockchain data. EveLog could be further refined regarding usability and performance by incorporating features like mapping log fields to multiple values. This enhancement would increase the expressiveness of the logs and enable more sophisticated data manipulation. Additionally, validating EveLog across various use cases from different sectors – such as healthcare, manufacturing, the Internet of Things (IoT), decentralized finance, metaverses, and

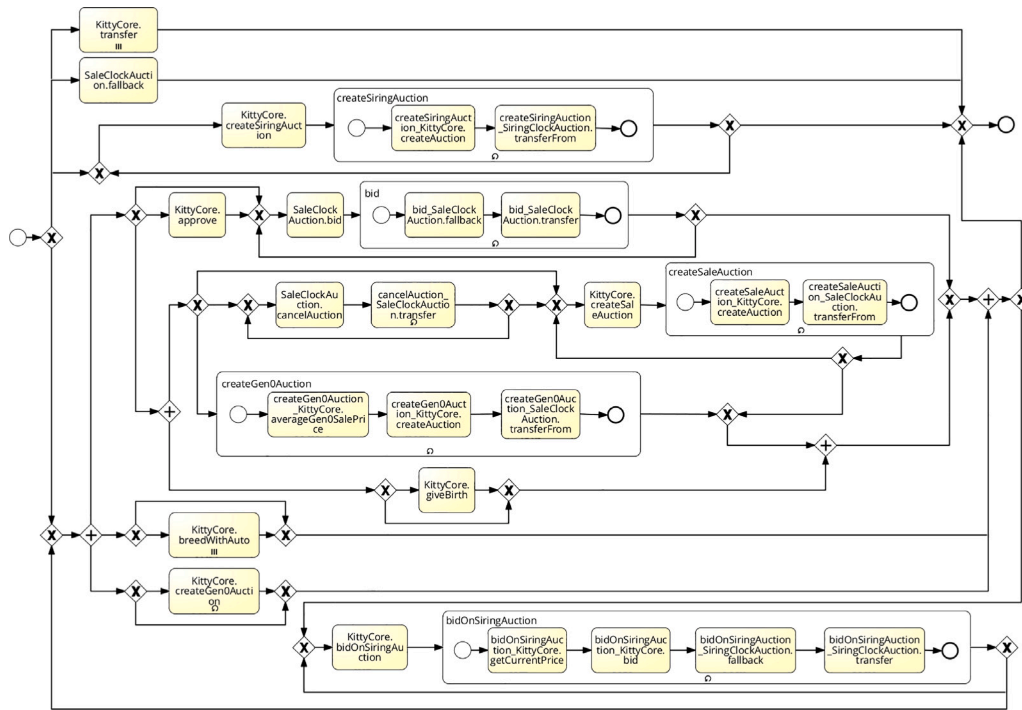


Fig. 10. Model discovered with $\text{BPMNMiner}_{\text{Inductive}}$.

blockchain-based games – would strengthen its generalizability beyond the CryptoKitties case study used in this research. We also suggest testing and evaluating the generated XES logs using other process discovery algorithms to identify the most accurate ones for blockchain-based applications. This might even lead to the development of a new process discovery algorithm optimized for blockchain environments. Lastly, future research could focus on analysing the execution of internal traces and exploring potential variations in the XES logs generated from them.

CRediT authorship contribution statement

Andrea Morichetta: Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Formal analysis, Conceptualization. **Yuri Paoloni:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Data curation, Conceptualization. **Barbara Re:** Writing – review & editing, Writing – original draft, Supervision, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, Engineering trustable and auditable choreography-based systems using blockchain, ACM Trans. Manag. Inf. Syst. 13 (3) (2022) 31:1–31:53, http://dx.doi.org/10.1145/3505225.

[2] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, F. Tiezzi, A flexible approach to multi-party business process execution on blockchain, Future Gener. Comput. Syst. 147 (2023) 219–234, http://dx.doi.org/10.1016/J.FUTURE.2023.05.006.

[3] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, A. Ponomarev, Caterpillar: A business process execution engine on the Ethereum blockchain, Softw. Pract. Exp. 49 (7) (2019) 1162–1193, http://dx.doi.org/10.1002/SPE.2702.

[4] L. Moctar-M'Baba, M. Sellami, W. Gaaloul, M.F. Nanne, Blockchain logging for process mining: a systematic review, in: International Conference on System Sciences, ScholarSpace, 2022, pp. 1–10.

[5] IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams, EEE Std 1849-2016 (2016), http://dx.doi.org/10.1109/IEEESTD.2016.7740858.

[6] C. Di Ciccio, G. Meroni, P. Plebani, Business process monitoring on blockchains: Potentials and challenges, in: Enterprise, Business-Process and Information Systems Modeling, in: Lecture Notes in Business Information Processing, 387, Springer, 2020, pp. 36–51, http://dx.doi.org/10.1007/978-3-030-49418-6_3.

[7] C. Di Ciccio, G. Meroni, P. Plebani, On the adoption of blockchain for business process monitoring, Softw. Syst. Model. 21 (3) (2022) 915–937, http://dx.doi.org/10.1007/S10270-021-00959-X.

[8] F. Corradini, A. Marcelletti, A. Morichetta, B. Re, A data extraction methodology for Ethereum smart contracts, in: IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events, PerCom 2024, 2024, pp. 524–529, http://dx.doi.org/10.1109/PERCOMWORKSHOPS59983.2024.10502604.

[9] R. Mühlberger, S. Bachhofner, C. Di Ciccio, L. García-Bañuelos, O. López-Pintado, Extracting event logs for process mining from data stored on the blockchain, in: C. Di Francescomarino, R. Dijkman, U. Zdun (Eds.), Business Process Management Workshops, Springer International Publishing, Cham, 2019, pp. 690–703, http://dx.doi.org/10.1007/978-3-030-37453-2_55.

[10] C. Klinkmüller, A. Ponomarev, A.B. Tran, I. Weber, W. van der Aalst, Mining blockchain processes: Extracting process mining data from blockchain applications, in: Business Process Management: Blockchain and Central and Eastern Europe Forum, Springer International Publishing, Cham, 2019, pp. 71–86, http://dx.doi.org/10.1007/978-3-030-30429-4_6.

[11] R. Hobeck, I. Weber, Towards object-centric process mining for blockchain applications, in: Business Process Management: Blockchain, Robotic Process Automation and Educators Forum - BPM 2023 Blockchain, RPA and Educators Forum, in: Lecture Notes in Business Information Processing, vol. 491, Springer, 2023, pp. 51–65, http://dx.doi.org/10.1007/978-3-031-43433-4_4.

[12] F. Corradini, F. Marcantoni, A. Morichetta, A. Polini, B. Re, M. Sampaolo, Enabling auditing of smart contracts through process mining, in: From Software Engineering To Formal Methods and Tools, and Back, in: Lecture Notes in Computer Science, vol. 11865, Springer, 2019, pp. 467–480, http://dx.doi.org/10.1007/978-3-030-30985-5_27.

[13] V. Buterin, Ethereum: A next-generation smart contract and decentralized application platform., 2014, https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf, [Online; Accessed 16 September 2022].

[14] W. Van Der Aalst, Process mining: Overview and opportunities, ACM Trans. Manage. Inf. Syst. (TMIS) 3 (2) (2012) 1–17, http://dx.doi.org/10.1145/2229156.2229157.

[15] A.J.M.M. Weijters, W.M. van der Aalst, A.K.A. de Medeiros, Process Mining with the Heuristicsminer Algorithm, Tech. rep., Technische Universiteit Eindhoven, ISBN: 978-90-386-0813-6, 2006.

[16] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Discovering block-structured process models from event logs - a constructive approach, in: J.-M. Colom, J. Desel (Eds.), Application and Theory of Petri Nets and Concurrency, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 311–329, http://dx.doi.org/10.1007/978-3-319-07734-5_6.

[17] R. Conforti, M. Dumas, L. García-Bañuelos, M. La Rosa, Beyond tasks and gateways: Discovering BPMN models with subprocesses, boundary events and activity markers, in: S. Sadiq, P. Soffer, H. Völzer (Eds.), Business Process Management, Springer International Publishing, Cham, 2014, pp. 101–117, http://dx.doi.org/10.1007/978-3-319-10172-9_7.

[18] I. Weber, C. Klinkmüller, H.M.N.D. Bandara, R. Hobeck, W.M.P. van der Aalst, Process mining on blockchain data: A case study of augur, in: Business Process Management, Springer International Publishing, Cham, 2021, pp. 306–323, http://dx.doi.org/10.1007/978-3-030-85469-0_20.

[19] P. Beck, H. Bockrath, T. Knoche, M. Digtiar, T. Petrich, D. Romanchenko, R. Hobeck, L. Pufahl, C. Klinkmüller, I. Weber, BLF: a blockchain logging framework for mining blockchain data, in: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track At BPM 2021, in: CEUR Workshop Proceedings, vol. 2973, CEUR-WS.org, 2021, pp. 111–115.

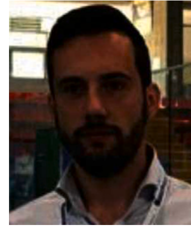
[20] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, A. Ponomarev, Caterpillar: A business process execution engine on the Ethereum blockchain, Softw. - Pract. Exp. 49 (2019) 1162–1193, http://dx.doi.org/10.1002/SPE.2702.

[21] A.B. Tran, Q. Lu, I. Weber, Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management, in: Proceedings of the Dissertation Award, Demonstration, and Industrial Track At BPM 2018 Co-Located with 16th International Conference on Business Process Management, BPM 2018, in: CEUR Workshop Proceedings, vol. 2196, CEUR-WS.org, 2018, pp. 56–60.

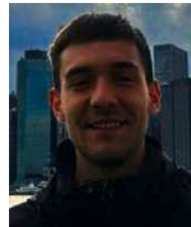
[22] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, J. Mendling, Untrusted business process monitoring and execution using blockchain, in: M. La Rosa, P. Loos, O. Pastor (Eds.), Business Process Management, Springer International Publishing, Cham, 2016, pp. 329–347, http://dx.doi.org/10.1007/978-3-319-45348-4_19.

[23] R. Jia, S. Yin, To EVM or not to EVM: blockchain compatibility and network effects, in: F. Zhang, P. McCorry (Eds.), Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security, DeFi 2022, ACM, 2022, pp. 23–29, http://dx.doi.org/10.1145/3560832.3563442.

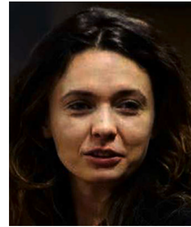
[24] M. Dumas, Constructing digital twins for accurate and reliable what-if business process analysis., Problems@ BPM 2938 (2021) 23–27.



Andrea Morichetta is an Associate Professor of Computer Science at the University of Camerino. He received his Ph.D. in Computer Decision and Systems Science from IMT School for Advanced Studies Lucca in 2016. His research interests are mainly in the area of formal methods and software engineering applied to distributed systems, business process management, and blockchain technologies. In the last few years, Morichetta has focused his interest on distributed ledger technologies and blockchain application development, with a particular focus on model-driven approaches.



Yuri Paoloni is a software engineer at Reply, an Italian consulting company. He graduated in Computer Science at the University of Camerino in 2022. His areas of work include large language models, digital humans, web development, mobile development, and backend development. His research interests are mainly in the field of process mining and blockchain technologies.



Barbara Re is an Associate Professor of Computer Science at the University of Camerino. She received her Ph.D. in Information Science and Complex Systems from the University of Camerino. Her research interests refer to the area of Business Process Management from modelling to analysis. Particular attention is paid to the use of formal methods as methodological and automatic tools for developing high-quality process-aware information systems. She was involved in multidisciplinary research projects collaborating with national and international research institutes and companies.