Computer Communications 240 (2025) 108204

Contents lists available at ScienceDirect

Computer Communications

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

A methodology for extracting and decoding smart contracts data

Flavio Corradini, Alessandro Marcelletti *, Andrea Morichetta, Barbara Re

University of Camerino, Via Madonna delle Carceri 7, Camerino, 62032, Italy

ARTICLE INFO

Keywords:
Blockchain
Data extraction
Ethereum
Smart contracts
State changes

ABSTRACT

Blockchain technology has been widely adopted to enhance the security and the decentralisation of smart applications in large-scale pervasive systems. In such a context, data extraction is crucial as it provides a better understanding of the system's behaviours. However, several challenges arise in automatically extracting data, due to the variety of data sources, such as transactions, events, contract storage, and the complexity of the blockchain structure. In particular, retrieving smart contract state changes remains unexplored despite its potential usage for discovering unexpected behaviour. For such reasons, in this work, we propose a novel methodology and a supporting application for extracting smart contract state changes and other execution-related data. The obtained data is then decoded and offered in a standard format to be easily reused. The methodology provides additional functionalities such as transaction filtering and capabilities for querying over extracted data. The effectiveness and the performance of the methodology were evaluated on three real-world projects from different EVM-based blockchains.

1. Introduction

Blockchain is gaining considerable attention to guarantee security and decentralisation in large-scale pervasive systems. Thanks to blockchain's security, it is possible to mitigate Distributed Denial-of-Service attacks [1] and secure communications between devices and users [2]. Its decentralised nature helps eliminate single points of failure associated with traditional single-server or third-party services [3,4]. In particular, blockchain provides novel solutions for the Internet of Things [5], Edge Computing [6], and 5G networks [7]. Moreover, adopting smart contracts in large-scale pervasive systems enables automated actions and encodes immutable conditions directly within the blockchain. In such a context, data generated from smart contracts can be used for certified auditing and monitoring activities [8–11], as well as new data analytics perspectives [12]. Indeed, smart contract analysis permits a better understanding of the system's behaviour by detecting anomalies, fraud, and vulnerabilities. Extracting data from executed smart contracts is crucial in supporting the analysis and the continuous improvement of blockchain applications. Data extraction is the most time- and effort-consuming task in an analysis project, typically requiring more than 80% of the resources [13]. For this reason, in this work we do not consider analysis activities but focus only on the extraction.

Data extraction from blockchain presents several challenges. The execution of a smart contract generates data stored in blocks (e.g., timestamps), transactions (e.g., sender, inputs, gas, and more), events, and storage (i.e., the memory containing the smart contract state) [14]. Additional effort is required to decode information that cannot be easily interpreted in its original form on the blockchain. Thus, the extraction activity must address the heterogeneity of storage and decoding factors. Moreover, catching a contract's state changes permits a comprehensive understanding of the application and enables detailed analysis of the contract's evolution over time. Differently from a transaction or a block, a state change does not generate a clear and accessible track, requiring a deep investigation of the low-level data structure [13,15]. In Ethereum-based blockchains, each variable influencing the state of a smart contract is permanently stored and encoded in the storage memory based on a specific slot. This slot is statically assigned for simple variables, while for complex types (e.g., mappings and structs), it is dynamically combined with a key generated during the execution. In the last few years, some approaches were proposed to extract data stored in different blockchain sources [16]. However, these approaches mainly extract information related to the execution of smart contract functions (e.g., events, inputs, senders) without considering the evolution of its state.

For these reasons, we propose **a data extraction methodology to extract data from smart contracts, including execution-related data and state changes**. To this aim, our methodology first captures knowledge about the contract transactions and extracts the related state changes for each of them. This is possible by replaying transactions inside the Ethereum Virtual Machine (EVM) and obtaining the traces

* Corresponding author.
E-mail address: alessand.marcelletti@unicam.it (A. Marcelletti).

https://doi.org/10.1016/j.comcom.2025.108204
Received 13 July 2024; Received in revised form 22 November 2024; Accepted 28 April 2025
Available online 12 May 2025

generated to reconstruct the history of changes in smart contract variables. Usually, this leads to exploiting archive nodes requiring a size of several TB of memory, depending on the client being used, and to define ad-hoc solutions with strong domain knowledge [13,15]. Our methodology relies on a resource-efficient solution in terms of used technologies (e.g., massive data storage) and information. Our proposal permits the extraction of traces without needing an archival node or other heavy data sources. To demonstrate the feasibility of the proposed solution, the methodology was implemented as a web application that extracts smart contract data according to user inputs. To enhance the efficiency of the methodology, data is stored in a database for faster retrieval in case of subsequent extractions. Furthermore, a dedicated query interface allows for defining complex queries leveraging a traditional Database Management System. The proposed methodology can be generally applied to any EVM-based smart contract, providing high compatibility with other blockchain networks. However, given the unique and different implementations of non-EVM solutions, the methodology currently does not aim to support these scenarios. To show the application in practice, we extracted data from three real-world projects on the Ethereum, Polygon and Fantom blockchains, measuring the related performance.

This paper extends the work initially proposed in [17] as follows.

- The methodology was extended by including two additional steps: the configuration for transaction filtering and the querying for retrieving extracted data.
- The extraction and decoding were extended to support the complete set of Solidity data types and their nested combination.
- The open-source web application was advanced to support the new steps of the methodology and its efficiency was improved by relying on a transaction history database.
- The validation was extended by including the performance analysis of three real-world projects from different EVM-based blockchains.
- The comparison with the current state of the art was advanced by including EVM data retrieval and analysis approaches.

The rest of the paper is organised as follows. Section 2 introduces the relevant characteristics of blockchain by focusing on Ethereum, representing the starting point for the proposed methodology and the foundation for the applicability to other EVM-based networks. Section 3 describes the proposed methodology, the different steps and the adopted solutions. Section 4 discusses the implemented application, while Section 5 shows the methodology at work on three real-world projects and its related performance. Finally, Section 6 provides an overview of related works, and Section 7 concludes the paper by pointing out future works.

2. Background

Ethereum is a decentralised, open-source blockchain with smart contract functionality [18]. Its public and permissionless characteristics permit participants to freely interact with it while ensuring that ledger data is accessible and visible to any interested party. Smart contracts are programs deployed on a blockchain which code is immutable and 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. In Ethereum, these contracts can be coded using the *Solidity* programming language,[1] which runs on the Ethereum Virtual Machine (EVM), similar to traditional programming languages. Once the code is generated, it is compiled into a low-level bytecode executed inside the EVM. After the smart contract is deployed in the blockchain through a dedicated transaction, it becomes available for user interaction. The compiled bytecode is executed as several EVM opcode instructions, which perform predefined operations deterministically. The EVM can store data inside the storage, memory, and the stack. Each smart contract has a data area called *storage*, a persistent key–value store between function calls and transactions. During its execution, the EVM performs low-level operations in a data area called *stack*. Furthermore, contracts can use a *memory* location, which is cleared for each message call. Each executed function of a smart contract can lead to an update of the global state of the ledger, maintained by nodes and containing information about balances and data. In a smart contract, the state variables are stored in the storage through slots, which are assigned to them depending on their size.

In Ethereum, transactions are cryptographically signed instructions sent on the blockchain by an account. *Public transactions* represent the transfer of cryptocurrency or the execution of a smart contract function. *Internal transactions* instead occur between smart contracts, lack a cryptographic signature, and are typically stored off-chain, meaning they are not a part of the blockchain. After the execution is completed, the transaction is added to a block and propagated in the network where its data is included such as *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. In addition, a transaction can include an additional log containing *events* emitted during the execution of smart contracts and include custom application data. Those events are used to log some custom information, and they are used to expose the outcome of an operation (e.g., transfer of a token, deposit, etc.). The resulting logged data is then used by external services, like front-ends, to update their internal states accordingly. Internal transactions occur instead between smart contracts, triggered when an external address calls a smart contract to execute an operation. The contract then uses its built-in logic to start interacting with the other required contracts 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.

3. The extraction methodology

This section describes the proposed methodology focusing on the challenges of extracting smart contract state changes. Fig. 1 depicts the steps of our proposal. The sequence of activities is managed using solid arrows while the used inputs and the produced outputs are connected to each activity through dashed arrows. The methodology starts by defining the initial configuration and filters used to retrieve the smart contract code and transactions. After compiling the contract, the extraction of data related to state changes, events, transactions, and blocks is conducted. Finally, the results are inserted into a log and saved on a local database enabling the user to the successive data querying. In the following, we analyse the entire methodology, providing detailed information on each step.

Configuration. The initial step of the methodology foresees the configuration of the fields identifying the contract from which to extract data. A first field refers to the *network* to use, such as a particular mainnet or testnet. A *block range* is also necessary to restrict the interval of transactions to retrieve. Some additional information can be set to filter transactions to extract in the next step. This is done by specifying fields related to *gas used*, *gas price*, *interval of time*, set of *sender addresses* and set of *executed functions*.

Get contract code. This step retrieves the target smart contract source code which is later compiled. To this purpose, the *contract address* and the *contract name* are taken as input. To provide a fully automated procedure, if the contract code is verified and publicly available, it is directly acquired, otherwise, the user can upload it manually.

[1] https://solidity.readthedocs.io/.

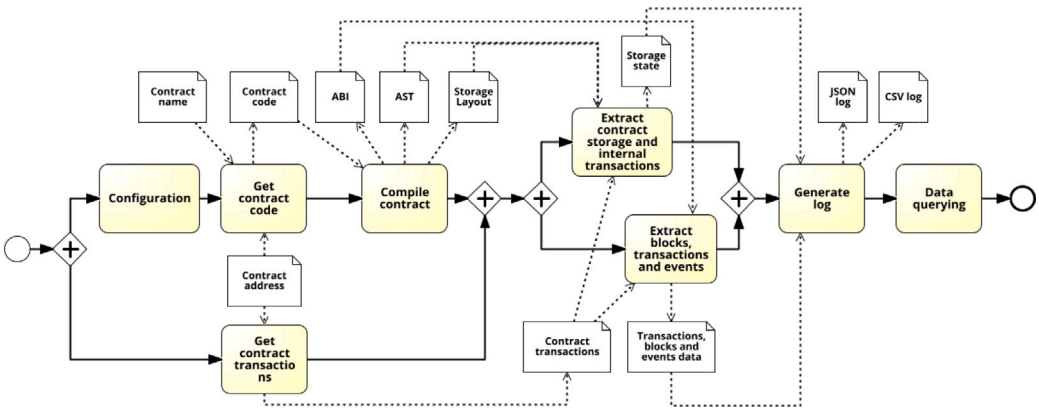


Fig. 1. Proposed data extraction methodology.

Get contract transactions. The scope of this step is to collect all the *transactions* referring to the specified smart contract. For this purpose, the list of transactions between the defined block interval is initially retrieved. Then, transactions are filtered according to the previously defined fields.

Compile contract. Once the smart contract source code is obtained, the Solidity compiler is used to get three particular outputs: (i) Application Binary Interface (ABI), (ii) Abstract Syntax Tree (AST), and (iii) storage layout.

The **Application Binary Interface** is a general-purpose data exchange format containing contract function types, names, inputs, outputs, and mutability information. Thanks to this interface, it is possible to interact with smart contracts from outside the blockchain and for contract-to-contract interaction. Indeed, inside the EVM, the deployed code is stored as bytecode and requires the ABI to interpret it. In this work, the ABI is primarily used to decode the information extracted from transactions and events since they are stored in the blockchain in a hexadecimal format. This information is, for instance, the name of the executed function, the inputs, and event values.

Listing 1: Highlight the AST with a focus on a variable assignment.

```
1  ...
2  'nodeType': "ContractDefinition",
3  ...
4  'nodes': [{
5      'id': <variableID>,
6      'name': "<variableName>",
7      'nodeType': "VariableDeclaration",
8      'stateVariable': <isStateVariable>,
9      'typeDescriptions': {...},
10     ...},
11     {
12     'name': "<functionName>",
13     'nodeType': "FunctionDefinition",
14     'statements': [
15         'expression': {
16             'leftHandSide': {
17                 'name': "<variableName>",
18             }
19             'nodeType': "Assignment",
20             'operator': "<operatorType>",
21             ...}]
22     }, ...]
```

The **Abstract Syntax Tree** of the smart contract is a tree representation of the source code in which each node contains a particular construct, such as a function and its statements. Inside the AST each smart contract function also contains the state variables it modifies, permitting the understanding of the scope of storage changes. By analysing the generated AST, the methodology associates the functions in the contract and the state variables they modify. This association is combined with the storage layout during the contract state extraction to recognise which state variables are updated in the executed functions. Listing 1 reports an extract of the AST structure referring to a smart contract with a state variable and a function modifying it. Generally, the AST starts with the representation of the smart contract definition node (Line 2). This contract node can contain in turn other nodes, such as those for the variable declarations and function definitions. In the example, the first node defines a state variable specifying its name and type (Lines 5–9). The second node shows a function definition along with its internal statements (Lines 12–21). Within this function node, there is an expression indicating the assignment of the variable *leftHandSide* (Lines 16–18) and the details of the operator involved (Lines 19–20).

Listing 2: Example of a storage layout template with a variable object.

```
23 'storage': [{
24     'astId': <astIdReference>,
25     'contract': "SolidityPathName:
           contractName",
26     'label': "<variableName>",
27     'offset': <offsetNumber>,
28     'slot': "<slotNumber>",
29     'type': "<variableType>"
30     }, ...]
```

The **Storage layout** interface contains information on how contract state variables are stored in the EVM storage memory. This memory is divided into 2^{256} contiguous slots of 32 bytes each in a key–value format and are referenced by indexes. Each state variable of a smart contract is mapped to a storage slot which index is assigned according to specific rules and optimisations. In this work, we refer to two kinds of variables: *simple* and *dynamic*. For simple state variables, the initially assigned slot index represents the storage's final position (i.e., the storage key). This is the case with statically sized variables such as uints, strings, arrays, etc. For dynamic variables instead, the storage key is generated by hashing the slot index concatenated with the dynamic data. This is the case of mappings and dynamic arrays, where new items are added to the variable at runtime. With this premise, the storage layout contains the default assignment of slot indexes for state variables made at compilation time. This information, combined with the observed changes in the storage at runtime, permits precise recognition and decoding of which state variable is effectively updated. However, this association is direct only for some simple variables, while optimisation rules and the generation of storage indices for dynamic variables pose a significant challenge. For this reason, in the case of simple variables, the storage layout is sufficient to derive their precise location directly. In contrast, in the case of dynamic variables, we use the storage layout

as a starting point to compute their final location. Listing 2 shows an example of the storage layout structure, containing the list of the contract state variables (Lines 23–30). Each variable representation contains general information such as the AST identifier (Line 24), the name of the reference contract (Line 25), the variable name (Line 26), the offset and the slot index (Lines 27–28), and the variable type (Line 29).

Extract contract storage and internal transactions. This step is the core point of the proposed methodology, and it extracts the state variables updated during each transaction found in the defined block range. For this purpose, each transaction is replayed in a local environment using the state of the blockchain at the moment in which the transaction was originally executed. This is done by fetching the state of the parent block from a blockchain node, which makes it available without the need to recompute it. Considering, for instance, the analysis of a contract C in the block range $[l, u]$, the state of C at block l is obtained directly from a node, without replaying all the previous transactions.

At this point, after the replay of the target transaction, a trace is returned, containing, among others, the list of executed operations (i.e., opcodes) and the state of the EVM (i.e., storage, stack and memory). In particular, the opcodes represent operations such as including a new variable or calculating a storage index. The EVM state contains instead different data such as the input/output of an operation or the content of the storage. This step reads the opcodes and the EVM state to reconstruct the state variable changes, matching this information to the storage layout and AST to identify precisely which state variable was updated, as well as in the case of dynamic ones. In the following, we describe the standard EVM opcodes that we consider in the proposed methodology and how they are used to find and decode state variables.

STOP indicates that the execution of a smart contract is terminated. This opcode contains the final updated contract storage slots showing the hexadecimal keys and values. Notably, these keys are computed with the Keccak256 of the storage slot index or the Keccak256 of the slot index concatenated with a dynamic value. In the case of simple variables, the storage key is a simple integer and its decoding returns the exact slot index. By reading this index in the storage layout it is possible to identify the original variable and decode its value. However, in the case of dynamic variables, the keys are more complex and additional information is required for their decoding.

SSTORE is the opcode operation that saves data inside the storage slots. Inside this opcode, the memory contains the slot keys and the pushed values, making it possible to catch each new insertion or update in the storage. During the extraction, we use this opcode to double-check whether a detected state variable change is valid and effectively related to a storage save.

KECCAK256 is the operation for the Keccak256 encryption. Among its various purposes, the KECCAK256 is also used to compute the storage key of dynamic state variables by hashing the default assigned storage slot index and the dynamic value. Both the index and the dynamic value are the inputs of the opcode and are visible in the EVM state along with the produced output. As output, the KECCAK256 generates the final variable storage key, corresponding to the one in the storage memory of the STOP opcode. By analysing KECCAK256 operations, it is therefore possible to catch the original storage slots index. At this point, reading the storage layout permits the identification of the precise state variable. This approach also works for nested variables, providing support for all possible cases.

CALL, DELEGATECALL, and STATICCALL are the opcodes referring to contract-to-contract executions producing the so called *internal transactions*. The main differences among these reside in the inputs and the execution context. In particular, the *CALL* executes a function code in another smart contract, modifying the target state. The *DELEGATECALL*, instead, invokes a function of another smart contract, keeping its original context (i.e., message sender and value) and modifying the original state. Finally, *STATICCALL* is used for read-only operations without changing any state. During the data extraction step, these operations are used to read internal transaction data such as the target contract address, the function selector (i.e., name in hexadecimal format), and the input parameters, providing additional details on the contract interactions.

A detailed visualisation of the state changes extraction and decoding is reported in Fig. 2. The first step concerns the extraction of data by replaying the transaction based on its hash. Once replayed, the execution trace is obtained and the state changes are retrieved from the storage returned in the STOP opcode. Such storage represents the updated state variables and is composed of several entries having a key (i.e., storage location) and a value (i.e., hex variable value). For each entry, the key is then checked. Since the key represents the storage slot, if it is a simple one (i.e., numeric), the state variable associated with it is identified, allowing for decoding of the raw value in the storage. In case instead, the storage key is a complex one (i.e., mapping case), the original slot has to be retrieved. This is done by analysing the list of KECCAK256 operations that calculate storage keys. The list is iterated until the desired storage slot is found. At this point, the state variable is identified and used to decode the storage value.

Extract blocks, transactions and events. Once the contract state changes and internal transactions are collected and decoded, the methodology reads information associated with transactions, blocks and events. For each transaction, the name of the executed function and its inputs are retrieved and decoded thanks to the ABI. Then, other attributes such as hash, sender, timestamp, and gas used are read. Finally, events emitted by the transactions are captured together with their names and the value of the attributes, decoded with the ABI.

Generate log. After all the previously mentioned data is extracted, this step generates the output log which is provided to the user. The log can be generated in JSON and CSV formats to support higher compatibility with modern analysis techniques. During this step, the extracted data is also saved in a local database, enabling the querying step and optimising successive extractions. Indeed, in case the same transactions are extracted twice, these are retrieved directly from the database. Listing 3 presents the JSON schema of the log with a focus on contract data and storage state. The log is structured into several main sections, each containing objects that refer to different types of information.

Listing 3: JSON schema for extracted data with a focus on contract and storage state.

```
{
  ...
  'items': {
    'type': "object",
    'properties': {
      'txHash': {'type': "string"},
      'blockNumber': {'type': "string"},
      'contractAddress': {'type': "string"},
      'sender': {'type': "string"},
      'gasUsed': {"type": "string"},
      'activity': {'type': "string"},
      'timestamp': {'type': "string"},
       ...
      'storageState': {
        'type': "array",
        'items': {
          'type': "object",
          'properties': {
            'variableId': {'type': "string"},
            'variableName': {'type': "string"
                },
            'type': {'type': "string"},
            'variableValue': {'oneOf':
            [{'type': "string"}, {'type': "
                integer"}]},
```

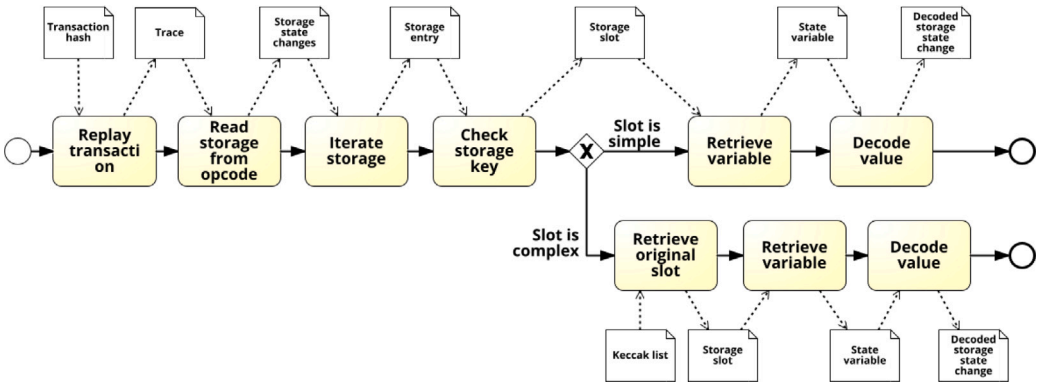


Fig. 2. Detailed extraction and decoding of storage state changes.

```
54                'variableRawValue': {'type': "
                      string"}
55              },
56            ...
57           }}
58         ...
59 }}
```

The first section (Lines 35–43) contains general information about the smart contract and the executed transaction, such as the sender address, executed activity, timestamp, and more. The second section includes the input object, which lists the names and values of all inputs passed to the function. Next, the log contains the storage state object (Lines 44–55). This section includes all state variables changed during the transaction, detailing their names, types, decoded values, and raw values containing the original hexadecimal values. Following the storage state, the internal transaction object provides details about internal calls made within the transaction, including the call type, target contract, and hexadecimal inputs. Finally, the event object identifies all events emitted during the transaction execution, including their names and values. To notice, for each described object, we assigned an ID composed of the object type, its occurrence index, and the transaction hash. This allows for the unique identification of information within each transaction, such as specific variables or events, to facilitate possible further analysis.

Data querying. In addition to the log, the methodology also provides a data querying step where the user can interact with the extracted data. Indeed, during the previous steps, such data is saved in a local database, accessible by the user with querying capabilities. In this way, the methodology permits faster data retrieval, without the need to replay transactions every time. Also, the usage of a standard DBMS permits the definition of complex queries and aggregation features.

4. Implemented application

In this section, we show the practical implementation of the methodology and all the steps described in Section 3. The methodology was implemented as a web application based on Node.js and its source code along with examples of logs are publicly available at https://pros.unicam.it/data-extraction-methodology/. To interact with the blockchain, the application relies on the Web3.js library[2] exploiting its functionalities for reading blocks, transactions, and events emitted from a contract. For the implementation of the contract code and transactions retrieval, the freely accessible Etherscan APIs[3] are used. The choice of using an external service simplifies the adoption of the methodology for users. Running a local node involves technical expertise for installation, updates, and synchronisation, as well as the resources needed to store data. In contrast, relying on an external service minimises the technical setup and maintenance required, making the methodology more accessible, even to non-expert users.

Configuration. To specify contract information, the application provides a user interface, as illustrated in Fig. 3, facilitating access to the methodology. The configuration page consists of two main panels. The first panel, depicted in Fig. 3(a), allows users to insert the fields about contract data such as the network, name, address, and block range. Currently, the application includes Ethereum, Polygon and Fantom blockchains, but it is extendable to any other EVM-based networks supported by the employed technologies. The second panel, shown in Fig. 3(b), is accessible by clicking on the top left yellow icon and contains the fields that filter the set of transactions to consider for the extraction. These fields include the range of gas used and gas prices, the interval of transaction timestamps, the list of sender addresses, and the list of executed functions.

Get contract code. After configuring contract information and fields, clicking the *Extract Data* button initiates the second step of the methodology. In this step, the application automatically calls the *getsourcecode* Etherscan API to search for publicly verified smart contract source code. Alternatively, users can upload the desired smart contract source code through the interface by providing the file in *.sol* format.

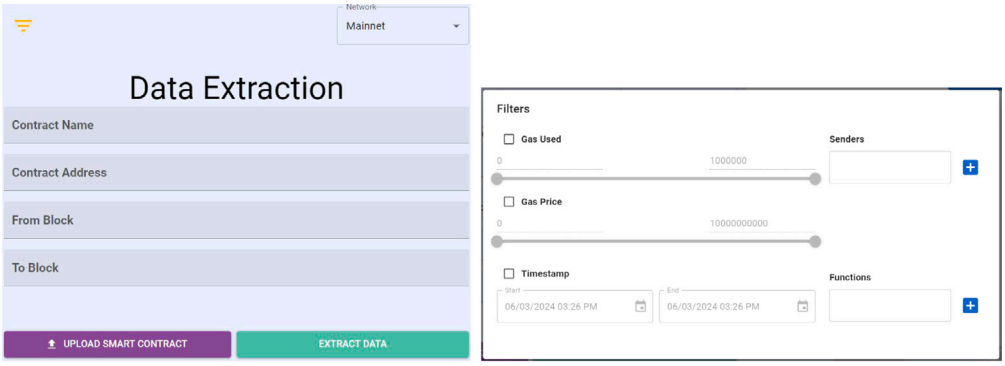
Get contract transactions. In this step, the application invokes the *txlist* Etherscan API to retrieve transactions associated with the identified smart contract address. The API returns all transactions within the specified block interval. The configured filtering fields are then applied to discard non-matching transactions.

Compile contract. The source code is compiled using Solc-js,[4] which requires a Solidity contract with a minimum version of 0.5.13 to produce the storage layout output. Whether the contract code is provided online or uploaded manually, the compilation returns the information described in the previous section. Notably, the application automatically detects the Solidity version and uses the appropriate Solc implementation. To provide concrete examples of the compilation outcomes, we provide in Listing 4 and Listing 5 two extracts from the AST and storage layout of the Ethereum PancakeSwap smart contract, used for the experimentation in Section 5.

Listing 4: Example of the AST for the PancakeSwap smart contract.

```
60 ...
61 'nodeType': "ContractDefinition",
62 ...
```

[2] https://web3js.readthedocs.io/en/v1.2.11/getting-started.html.
[3] https://docs.etherscan.io/api-endpoints/contracts.
[4] https://github.com/ethereum/solc-js.



(a) Panel for general contract data. (b) Panel for filter data.

Fig. 3. Configuration page.

```
63 'nodes': [{
64     'id': 2284,
65     'name': "defaultFeeBp",
66     'nodeType': "VariableDeclaration",
67     'stateVariable': true,
68     'typeDescriptions': {...},
69     ...},
70 {
71     'name': "setDefaultFeeBp",
72     'nodeType': "FunctionDefinition",
73     'statements': [
74         'expression': {
75             'leftHandSide': {
76                 'name': "defaultFeeBp",
77             }
78             'nodeType': "Assignment",
79             'operator': "=",
80             ...}
81     ]}, ...]
```

The AST shows the nodes defining a state variable named *defaultFeeBp*, having 2284 as ID (Lines 64–68), which is modified by the *setDefaultFeeBp* function (Lines 71–72). This function updates the variable through an *assignment* operation using the = operator (Lines 74–80). In contrast, the storage layout shows the *defaultFeeBp* variable, as part of the *CakeOFT* contract (Lines 83–84) and its slot assignment at index 8 (Lines 85–88).

Listing 5: Example of the storage layout for the PancakeSwap smart contract.

```
82 'storage': [{
83     'astId': 2284,
84     'contract': "contracts/eth/OFT.sol:
           CakeOFT",
85     'label': "defaultFeeBp",
86     'offset': 0,
87     'slot': "8",
88     'type': "t_uint16"}, ...]
```

Extract contract storage and internal transactions. To replay transactions locally and obtain traces, the application uses the Hardhat framework and the *debug_traceTransaction* functionality.[5] Hardhat requires a configuration file specifying the chain ID, block number, and network to fork. Based on the information previously provided by the user, the application automatically generates this configuration, simplifying the process. As a network provider, we rely on Alchemy[6] but other compatible providers can be used. After replaying the transaction, the EVM traces are obtained and processed as described in Section 3. This process results in the extraction and decoding of storage state changes and internal transactions.

Extract blocks, transactions and events. While reading the contract state and internal transactions, the application extracts blocks, transactions, and events. This includes retrieving data such as the sender, gas used, event values, and more. Using the ABI, the raw values are then decoded into a human-readable format.

Generate log. After all the extraction steps are completed, the resulting log is created and displayed to the user in a dedicated panel, as shown in Fig. 4. This step provides the user with an aggregated log of all extracted data in either JSON or CSV format, described as follows. To show the outcome of the extraction methodology, Listing 6 reports an example of the JSON log containing one transaction of the PancakeSwap smart contract.

Listing 6: Example of log entry for a single transaction.

```
89 'txHash':"0x2e10395b...6150b91225",
90 'contractAddress':"0x152649ea...ead6d4c898",
91 'sender':"0x48e4eb93...d6be5562f3",
92 'gasUsed': "230201",
93 'activity': "sendFrom",
94 'timestamp': "2024-05-13T13:45:47.000Z",
95 'inputs': [
96     1: { 'inputId':"event_62_0...6150b91225",
97          'inputName':"_dstChainId",
98          'type':"uint16",
99          'inputValue':102 } ... ],
100 'storageState': [
101     0:{ 'variableId': "variable_d...
            EAD6d4c898",
102         'variableName': "defaultFeeBp",
103         'type': "t_uint16",
104         'variableValue': "0",
105         'variableRawValue': "
              0000000000...000000000" } ... ],
106 'internalTxs': [
107     0: { 'callId': "call_42192...6150b91225",
108          'callType': "CALL",
109          'to': "0000000000...8401a178e2",
110          'inputsCall': [
111              0: "4d3a0f7c00...26eb48f7ea" ]}
                     ... ],
112 'events': [
```

[5] https://hardhat.org/hardhat-network/docs/overview#the-debug-tracetransaction-method.

[6] https://www.alchemy.com/.

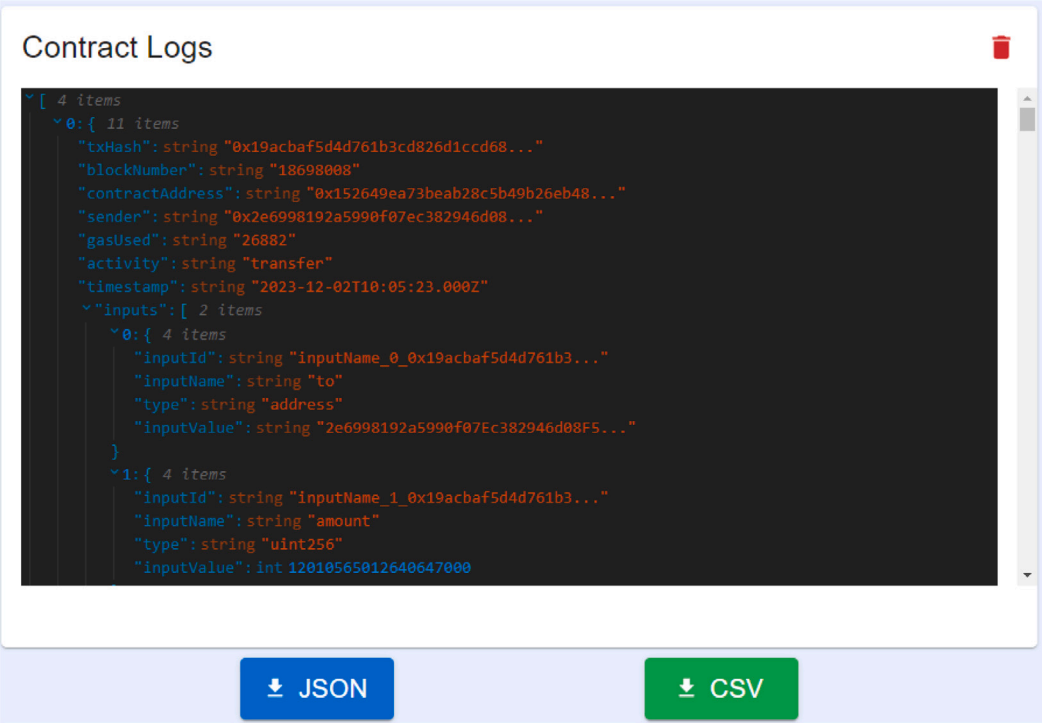


Fig. 4. Panel with the produced log.

```
    0: { 'eventId':"event_62_0...6150b91225",
         'eventName':"Transfer",
         'eventValues': {
             'from':"0x48e4eb93...d6bE5562F3"
                 ,
             'to':"0x00000000...0000000000",
             'value':876327370800000000 }
                 ... ]
```

Data querying. The final step of the methodology involves querying the extracted data stored in a local database during the previous steps. This allows users to directly query the data without needing to replay the transactions every time. The usage of a traditional DBMS also permits the definition of complex queries, enabling the aggregation of data from different smart contracts. Fig. 5 shows the dedicated query page. Here, the user can specify information about a single transaction or a set of transactions by selecting various fields. These fields include transaction hash, contract address, sender, gas used range, activity, block numbers, and timestamps. Additionally, users can apply additional fields related to the log objects previously described (i.e., inputs, storage state, events and internal transactions) to search particular values or types of data. Once the query is completed, all corresponding transactions are displayed in the right side panel and can be downloaded as a JSON file.

5. Methodology at work

To demonstrate the feasibility of the proposed methodology, in this section we report the performance analysis of the data extraction on three real-world projects from different domains and blockchains. In particular, we consider the PancakeSwap, Oval3 and FTM Fantom Miner applications from the Ethereum, Polygon and Fantom blockchains, respectively.[7]

As the experimental environment, we used a machine equipped with a processor Intel(R) Core(TM) i7-8750H CPU@2.20 GHz with 6 cores and 16 GB of RAM running Windows as the operating system. For each application, we extracted data from two samples of one hundred and one thousand transactions, respectively. With the first experimentation, we provide a perspective on the extraction performance and details about each scenario. With the second one, we compare the various trends to provide an evaluation also on the scalability of the methodology. A key parameter in this evaluation is the time required for transaction replay, as it represents the most time-intensive step.

Indeed, when considering a transaction, various factors affect its execution time during the replay. One of the most crucial factors measuring the single transaction complexity in EVM-based systems is the *gas used*, which indicates the computational work required to execute operations on the network and determines the transaction fee for processing and validation. However, during the replay of a transaction, all preceding transactions in the block are executed sequentially until the target transaction is reached. Consequently, understanding the complexity of replaying requires considering the cumulative gas used by all prior transactions in the block, as this directly impacts the computational effort required. From now on, we refer to this cumulative value as *total gas used*. For this reason, in the performed experiments we aim at providing insights into replaying performance and possible factors influencing it.

PancakeSwap. The first analysed scenario is the PancakeSwap decentralised application. PancakeSwap is the leading decentralised exchange based on the BNB Smart Chain with cross-chain bridges for token sharing. In particular, we considered the Ethereum smart contract[8] implementing the Cake token used for cross-chain capabilities. The extracted transactions were obtained from a block range between 19,858,013 and 19,868,214. Fig. 6 illustrates the general trend with the *total gas used* of each replayed transaction on the x-axis and the

[7] The full logs of the extracted smart contracts are available at https://pros.unicam.it/data-extraction-methodology/. We selected these scenarios as they represent popular cases in the respective platforms.

[8] https://etherscan.io/token/0x152649eA73beAb28c5b49B26eb48f7EAD6d4c898#code.

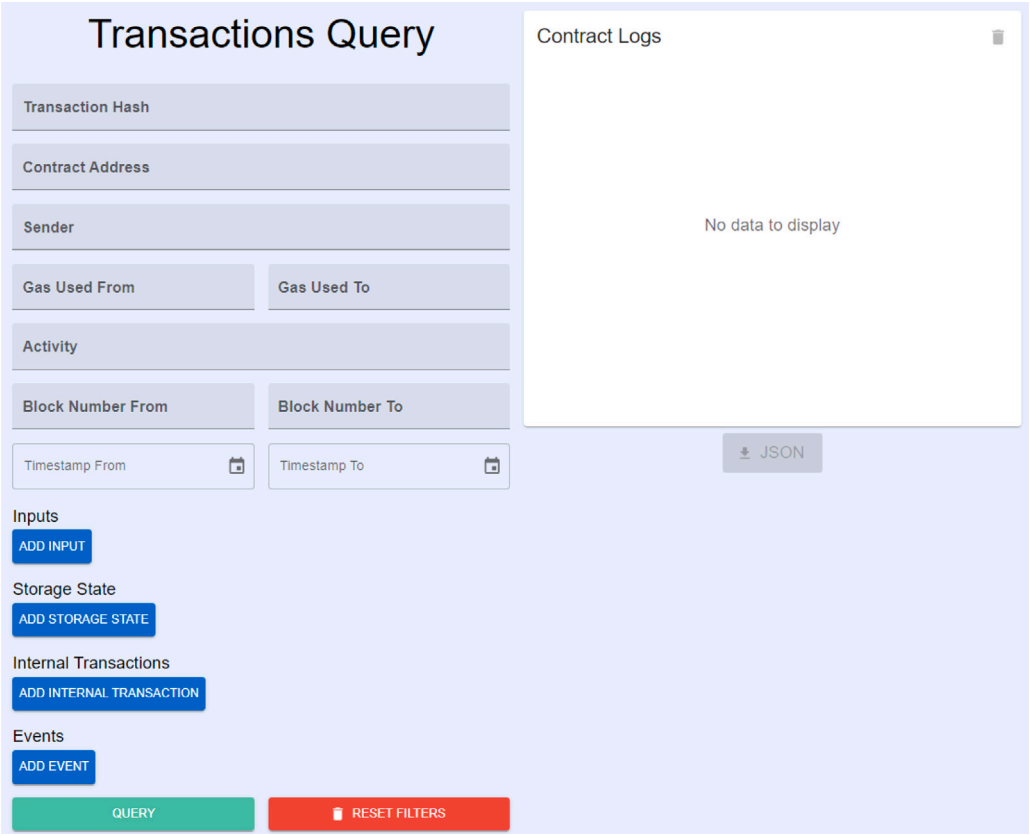


Fig. 5. Page for querying transactions from local database.

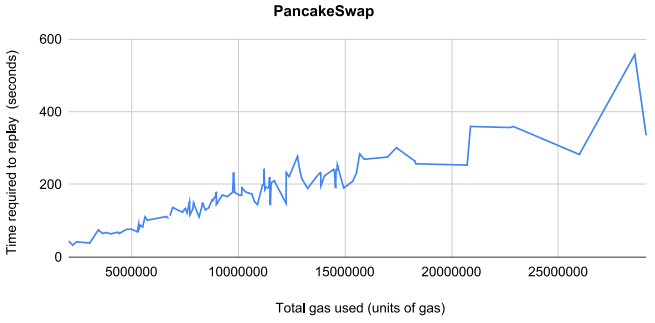


Fig. 6. Performance of replaying 100 transactions from PancakeSwap smart contract.

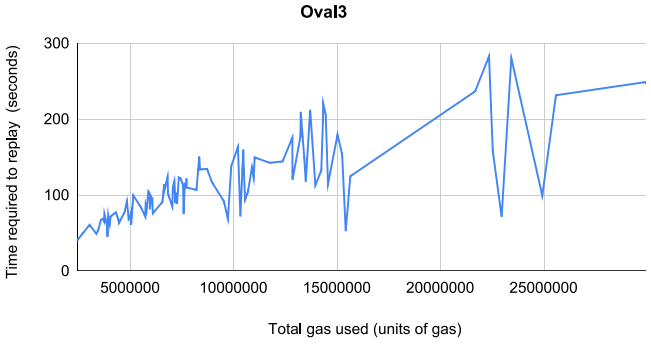


Fig. 7. Performance of replaying 100 transactions from Oval3 smart contract.

time needed to replay it in seconds on the *y*-axis. Notice, the decoding of storage state and other data required an average of 0.2609 ms for transaction thus not influencing the overall performance. Despite the chart showing high variability, it is possible to identify an upward trend, correlating the increase of total gas used to the replay time. The minimum time was 32 s for a transaction having a total gas used of 2,060,378 units of gas while the maximum was 557 s for a transaction with 25,980,830 total gas used. The average time was 178 s, with an average gas usage of 10,771,376 units.

Oval3. The second scenario considers the Oval3 application on the Polygon blockchain.[9] Oval3 is a fantasy game app on Rugby allowing users to play with official digital cards of professional players. The extraction results are reported in Fig. 7. The decoding time required an average of 0.1803 ms and it was not considered in the results. The transactions were obtained from a block range between 63,229,164 and 63,318,249. Also in this case, the results show high variability, with an upward trend that can be identified correlating extraction time and total gas used. Indeed, the minimum time required is 40 s to extract data from a transaction having a total gas used of 2,431,825. The maximum time is instead 282 s for a transaction with 29,932,283 of total gas used. The average is 114 s for gas usage of 9,500,742 units.

FTM Fantom miner. The third scenario is on a high-risk application on the Fantom blockchain, named FTM Fantom Miner.[10] The application permits the hiring of miners and the provision of a daily amount of FTM tokens. The extraction performance is reported in Fig. 8 and it excludes

[9] https://polygonscan.com/address/0x83a5564378839eef0721bc68a0fbeb92e2de73d2#code.

[10] https://ftmscan.com/address/0x6f123c6347521325d3a6cf08517a73bd1d64f191#code.

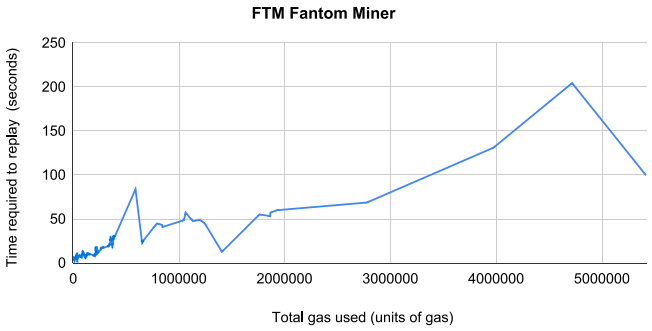


Fig. 8. Performance of replaying 100 transactions from FTM Fantom Miner smart contract.

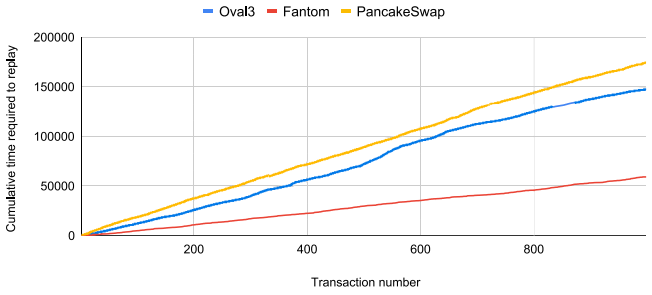


Fig. 9. Time performance scalability evaluation on 1000 transactions.

the decoding step, having an average of 0.2961 ms. The transactions were obtained from a block range between 87,409,147 and 88,741,148. Different from the previous cases, the extraction is more efficient while the upward trend and correlation between time and total gas used can still be confirmed. Indeed, the minimum extraction time took 2.85 s for a total gas used of 0 as it was the first transaction in the block. The maximum needed 204 s for a transaction of 5,412,922 total gas used. Finally, the average time is 17,6 s with a gas usage of 4,015,19 units.

Scalability evaluation. To provide an evaluation of the methodology scalability, in the second experimentation we considered a sample of one thousand transactions for each smart contract gauging the obtained trends in terms of time and gas. Fig. 9 contains the first trend, reflecting the cumulative time performance for the transaction replaying. This measure is the sum of the single replaying times for the one thousand transactions. As a result, a linear upward trend is obtained for all smart contracts, with some differences in their values. Indeed, the replay of 1000 transactions from the Pancake smart contract required a cumulative amount of 174,369 s. Similarly, in the Oval3 case, a cumulative of 147,740 s were needed while only 59,241 s in FTM Fantom Miner.

A different perspective is provided in Fig. 10. Here, the trend considers the incremental sum of the total gas used in each replayed transaction. The results show a linear upward trend similar to the previous case, with some differences in the amount of gas value. Indeed, during the replay of the transactions sample from the different smart contracts, a cumulative total of 10,130,431,082 gas units was consumed by PancakeSwap, 9,660,308,718 by Oval3 and 2,161,439,914 by FTM Fantom Miner.

Discussion. Observing the results of the experiments, the upward trends properly reflect the behaviour of the replay functionality. The initial experiments provided detailed measurements of the replay step, establishing a correlation between the total gas used and the required execution time. This correlation was confirmed in the scalability evaluation, where a larger sample of transactions demonstrated the relationships between the total gas used and the time. In both experiments, some

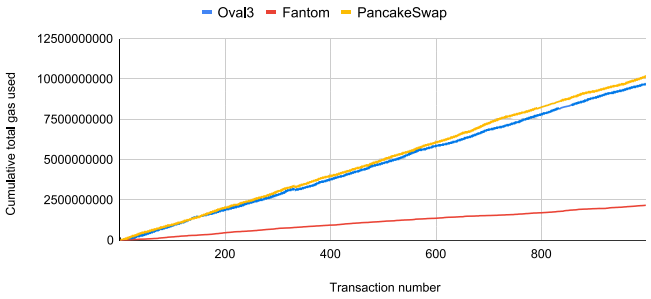


Fig. 10. Cumulative gas trend on 1000 transactions.

variations were evident. The causes of these variations may depend on very different aspects such as the number of executed opcodes, the used technologies, and the network provider. For example, in the extraction from the Fantom blockchain, it is evident that the differences in the underlying protocol influence the performance. Fantom is indeed designed to achieve high scalability [19] thus leading to the local transaction replay efficiency. This behaviour was confirmed in both experiments, where the trend remained linear but with more efficient performance. Furthermore, the selected sample of transactions is included in a large range of blocks, having different execution contexts and complexities, thus making precise measurements challenging. However, the overall trend is encouraging as it shows the practical applicability of the proposed methodology. In the future, we intend to explore other different technologies and understand their possible benefits in terms of performance.

To improve the efficiency of the extraction step, after transactions are retrieved from the blockchain, their data is saved in a database, storing a local copy of it. In this way, during successive executions, already extracted data is provided directly from the database, avoiding useless interactions with the blockchain and drastically reducing the time required. Indeed, the retrieval of a transaction from the database takes an average of 2.5 ms from the three extracted smart contracts, demonstrating its efficiency.

6. Related works

Data extraction for smart contract analysis. When analysing smart contracts, the replay of transactions and extraction of EVM traces play a crucial role, as they permit to obtain a detailed vision of the EVM operations. This technique was used in the literature to provide solutions enabling different kinds of analysis. In [20] the authors propose XBlock-ETH, a blockchain data analytics framework that extracts blockchain data such as blocks, traces and transaction receipts. In particular, XBlock-ETH reads opcodes from the replicated transactions to obtain different datasets for analysis and statistics. Ponzi schemes are instead detected using the proposed machine learning method in [21]. To this aim, opcodes are extracted from various smart contracts and used to build a random forest model used to detect Ponzi schemes. The TxSpector analysis framework for Ethereum transactions is proposed in [22] to identify smart contracts attacks and vulnerabilities to enable their forensics analysis. It replays transactions and takes opcodes storing them on a local database for optimisation. At this point, execution flows and logic relations are built and detection rules are applied to identify attacks. In [23] the Smartmuv tool for storage state extraction and upgrade of Ethereum smart contracts is presented. Smartmuv extracts contract storage for both simple and complex state variables by proposing a key approximation algorithm. This algorithm first calculates the storage slots by navigating the AST and identifying all declarations of state variables. Then, the smart contract source code is inspected to approximate the set of keys for mappings. The storage state is then extracted and migrated to the new upgraded contract.

Table 1
Table of identified related works and their characteristics. B = Blocks, CC = Contract Creations, CS = Contract State, E = Events, IT = Internal Transactions, T = Transactions.

Work	Target technology	Contract type	Considered data						Output format
			B	CC	CS	E	IT	T	
[24]	Ethereum	Process based	✓	✗	✗	✗	✗	✓	XES
[10]	Ethereum	Generic	✗	✗	✗	✗	✗	✓	XES
[25]	Ethereum	Generic	✓	✗	✗	✓	✗	✗	XES
[26]	Ethereum	Generic	✓	✗	✗	✓	✗	✗	XES
[27]	Ethereum, Hyperledger Fabric	Generic	✓	✗	✗	✓	✗	✗	XES
[28]	Hyperledger Fabric	Process based	✗	✗	✗	✓	✗	✗	CSV
[14]	Ethereum	Generic	✓	✗	✗	✓	✗	✗	ACEL
[29]	Ethereum	Generic	✓	✓	✗	✓	✓	✗	OCEL
This work	Ethereum, Polygon, Fantom	Generic	✓	✗	✓	✓	✓	✓	JSON

As highlighted in the presented works, traces and related opcodes are particularly useful during the analysis of smart contracts to detect vulnerabilities and other behaviours such as execution flows and scams. Extracting such data and providing it in a reusable manner is therefore crucial. Compared to the aforementioned works, our methodology presents some core differences. From a general perspective, the presented work does not aim to provide any specific analysis but focuses on the extraction phase. In the future, we can consider the integration of specific analyses, such as vulnerabilities or particular behaviours, to enhance the scope of the methodology and adapt it to more specialised use cases. Another difference regards specific data types. Our methodology is not limited to extracting low-level opcodes and raw data but conducts additional processes to decode data, presenting it in a high-level, human-readable format. For smart contract storage, differently from [23], we precisely identify and extract storage locations, associating them with the exact state variables and decoding their values. Finally, the implementation of a user-friendly application and the low-technical requirements represent other significant contributions.

Data extraction for process mining analysis. The employment of automatic methodologies for the extraction of blockchain data has experienced many proposals over the years. Particular interest was encountered in process mining, where automatic data extraction from execution logs is crucial to analysing the history of smart contracts. In this context, the available solutions start with the extraction of data mainly related to contract execution [16]. A first attempt was proposed in [24] to discover business processes executed on the Ethereum blockchain. In this framework, an XES log is derived from data related to blocks and transactions. Similarly, another early research described in [10] extracts Ethereum transactions and generates XES logs for process discovery and conformance checking analysis. In [25,26] instead, events emitted by Ethereum smart contracts are extracted and formatted according to XES for process discovery. The approach was then extended in [27], including the Hyperledger Fabric blockchain support. Fabric was also used in [28], where various data such as events, assets, transactions, and participants were obtained from the vehicle manufacturing network smart contract, resulting in a final CSV file. Recent works have also investigated the use of object-oriented standards. The Artifact-Centric Event Log (ACEL) logging format was proposed in [14]. ACEL extends the OCEL standard with concepts related to lifecycle, object changes, relations, and relation changes. The proposed extraction method turns in this direction by mapping Ethereum smart contract events into ACEL elements. Similarly, [29] extracts contract creation and message call data from Ethereum transaction traces that are used with events to generate OCEL logs. In this case, no extensions to the standard were applied, and the authors also included internal transactions, increasing the variety of data considered.

A resuming and comparison of the data extraction approaches is reported in Table 1, and it considers the (i) target technology, the (ii) contract type, the (iii) extracted data, and the (iv) output format. Regarding the first aspect, it is evident that Ethereum is the most used blockchain, while only a few works face a different technology, specifically Hyperledger Fabric [27,28]. However, while the described Ethereum-based approaches could be applied to other EVM platforms, they do not explicitly implement and test it. With our methodology instead, we intend to provide a general solution, integrating and supporting other EVM-based blockchains. The inclusion and extensibility represent core aspects of the proposed methodology. As application types, some works assume smart contracts representing business processes instances [24,28], thus having a more customised solution, while the rest of the approaches propose the extraction of data starting from generic applications. Also, in this work, we do not focus on a specific application type but remain as general as possible to support many different contexts. Another important aspect relates to the data sources, usually transactions, blocks, and events with their respective decoding. Transaction traces are used in [29] to get message calls and contract creations. Unlike the largely considered data sources, we use transaction traces to get smart contract state changes and internal transactions. To read the contract state, we dynamically build the storage locations that save state variables during a function execution. The remaining part of the table shows that all the approaches also decode the extracted data and that the well-established XES standard is used for the earliest work. With the recent growth around object-centric techniques, the OCEL standard has started to be considered [29], and an extension was also provided in [14]. In this work, we do not extract data based on a particular standard within its boundaries but aim to keep the extraction phase as a general step reusable in different contexts. For this reason, as output, we provide a general JSON log that can be easily adapted to any desired standard. The analysis of specific process-based cases, along with the adoption of process mining standards, offers an interesting area for further research. This, combined with the inclusion of additional data (e.g., contract creations of [29]) can provide novel and significant perspectives during analysis activities.

7. Conclusions

Blockchain is becoming crucial in securing communications and removing centralised computation in modern pervasive systems. From IoT to edge computing and 5G networks, smart contracts implement decentralised logic and manage access control rights and identities. Thanks to the transparency and immutability of the blockchain, data generated by smart contracts is open to innovative analysis techniques. By extracting data resulting from smart contract execution, it is possible to conduct analysis activities that enable the usage of such data to identify anomalies, vulnerabilities and unexpected behaviour. However, when applying analysis techniques to blockchain, different data sources are involved like events, blocks, and transactions, hence requiring novel solutions for their retrieval. In particular, the processing of smart contract state changes is still an open challenge due to the peculiarity of the blockchain structure. Reading state changes permits the consideration of a larger amount of information and represents behaviours that are not always expressed in blockchain logs.

Over the years, methodologies were proposed to automatically extract blockchain data and analyse it, eventually converting it to a particular standard. In this context, the extraction of smart contract state changes remains largely unexplored especially when this data has

to be decoded. For this reason, in this work, we present a data extraction methodology for EVM-based smart contracts dealing with the challenges of state extraction. For each generated transaction, the state of the contract and other execution-related information are extracted and decoded. Extracted data is then provided in a log with a general-purpose representation. In this way, it can be converted according to a standard depending on different needs and enabling further analysis. To optimise performance and availability, a local database maintains extracted data, permitting additional filtering with query functionalities. The methodology was implemented as a web application, providing a lightweight solution and abstracting from complexity, permitting its usage also to non-technical users. The methodology was tested over three real-world blockchain projects from different blockchains, measuring the extraction performance in different scenarios.

While the methodology currently supports EVM-based platforms, its applicability to different kinds of blockchains remains an open area for exploration. Solutions such as Hyperledger Fabric and Algorand present significant differences in their implementation, particularly in the state management and data structures. These differences prevent the direct application of the methodology, highlighting the need for targeted adaptations in future enhancements. In future works, we plan to extend the methodology to investigate contracts to contract executions and deal with complex proxy patterns. These patterns divide smart contract functionalities requiring further support to reconstruct the entire history of a transaction. Finally, we plan to provide a public collaborative repository where extracted datasets can be imported, analysed and made accessible to anyone interested.

CRediT authorship contribution statement

Flavio Corradini: Supervision, Resources, Funding acquisition, Conceptualization. **Alessandro Marcelletti:** Writing – original draft, Visualization, Validation, Software, Methodology, Conceptualization. **Andrea Morichetta:** Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Conceptualization. **Barbara Re:** Writing – review & editing, Supervision, Project administration, Methodology, 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.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Data availability

All data is publicy available with the links provided in the manuscript.

References

[1] R. Chaganti, B. Bhushan, V. Ravi, A survey on blockchain solutions in ddos attacks mitigation: Techniques, open challenges and future directions, Comput. Commun. 197 (2023) 96–112, http://dx.doi.org/10.1016/J.COMCOM.2022.10.026.

[2] S. Aggarwal, R. Chaudhary, G.S. Aujla, N. Kumar, K.R. Choo, A.Y. Zomaya, Blockchain for smart communities: Applications, challenges and opportunities, J. Netw. Comput. Appl. 144 (2019) 13–48, http://dx.doi.org/10.1016/J.JNCA.2019.06.018.

[3] M. Kara, H.R.J. Merzeh, M.A. Aydin, H.H. Balik, VoIPChain: A decentralized identity authentication in voice over IP using blockchain, Comput. Commun. 198 (2023) 247–261, http://dx.doi.org/10.1016/J.COMCOM.2022.11.019.

[4] X. Xu, Y. Guo, Y. Guo, Fog-enabled private blockchain-based identity authentication scheme for smart home, Comput. Commun. 205 (2023) 58–68, http://dx.doi.org/10.1016/J.COMCOM.2023.04.005.

[5] X. Wang, X. Zha, W. Ni, R.P. Liu, Y.J. Guo, X. Niu, K. Zheng, Survey on blockchain for internet of things, Comput. Commun. 136 (2019) 10–29, http://dx.doi.org/10.1016/J.COMCOM.2019.01.006.

[6] R. Yang, F.R. Yu, P. Si, Z. Yang, Y. Zhang, Integrated blockchain and edge computing systems: A survey, some research issues and challenges, IEEE Commun. Surv. Tutor. 21 (2) (2019) 1508–1532, http://dx.doi.org/10.1109/COMST.2019.2894727.

[7] D.C. Nguyen, P.N. Pathirana, M. Ding, A. Seneviratne, Blockchain for 5G and beyond networks: A state of the art survey, J. Netw. Comput. Appl. 166 (2020) 102693, http://dx.doi.org/10.1016/J.JNCA.2020.102693.

[8] C. Di Ciccio, G. Meroni, P. Plebani, Business process monitoring on blockchains: Potentials and challenges, in: Enterprise, Business-Process and Information Systems Modeling, in: LNBIP, vol. 387, Springer, 2020, pp. 36–51.

[9] 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.

[10] F. Corradini, F. Marcantoni, A. Morichetta, A. Polini, B. Re, M. Sampaolo, Enabling auditing of smart contracts through process mining, From Softw. Eng. Form. Methods Tools, Back 11865 (2019) 467–480, http://dx.doi.org/10.1007/978-3-030-30985-5_27.

[11] 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.

[12] A.I. Sanka, M. Irfan, I. Huang, R.C.C. Cheung, A survey of breakthrough in blockchain technology: Adoptions, applications, challenges and future research, Comput. Commun. 169 (2021) 179–201, http://dx.doi.org/10.1016/J.COMCOM.2020.12.028.

[13] K. Diba, K. Batoulis, M. Weidlich, M. Weske, Extraction, correlation, and abstraction of event data for process mining, WIREs Data Min. Knowl. Discov. 10 (3) (2020) e1346, http://dx.doi.org/10.1002/WIDM.1346.

[14] L. Moctar-M'Baba, N. Assy, M. Sellami, W. Gaaloul, M.F. Nanne, Process mining for artifact-centric blockchain applications, Simul. Model. Pr. Theory | J. 127 (2023) 102779, http://dx.doi.org/10.1016/j.simpat.2023.102779.

[15] J.D. Weerdt, M.T. Wynn, Foundations of process event data, in: Process Mining Handbook, in: LNBIP, vol. 448, Springer, 2022, pp. 193–211, http://dx.doi.org/10.1007/978-3-031-08848-3_6.

[16] 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, URL http://hdl.handle.net/10125/80091.

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

[18] V. Buterin, Ethereum: A next-generation smart contract and decentralized application platform, 2014, https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[19] F. Foundation, FANTOM whitepaper, 2018, https://fantom.foundation/_next/static/media/wp_fantom_v1.6.39329cdc5d0ee59684cbc6f228516383.pdf.

[20] P. Zheng, Z. Zheng, J. Wu, H. Dai, XBlock-ETH: Extracting and exploring blockchain data from ethereum, IEEE Open J. Comput. Soc. 1 (2020) 95–106, http://dx.doi.org/10.1109/OJCS.2020.2990458.

[21] W. Chen, Z. Zheng, E.C. Ngai, P. Zheng, Y. Zhou, Exploiting blockchain data to detect smart ponzi schemes on ethereum, IEEE Access 7 (2019) 37575–37586, http://dx.doi.org/10.1109/ACCESS.2019.2905769.

[22] M. Zhang, X. Zhang, Y. Zhang, Z. Lin, TXSPECTOR: uncovering attacks in ethereum from transactions, in: Security Symposium, USENIX, 2020, pp. 2775–2792.

[23] M. Ayub, T. Saleem, M.U. Janjua, T. Ahmad, Storage state analysis and extraction of ethereum blockchain smart contracts, ACM Trans. Softw. Eng. Methodol. 32 (3) (2023) 57:1–57:32, http://dx.doi.org/10.1145/3548683.

[24] 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: Business Process Management Workshops, in: LNBIP, vol. 362, Springer, 2019, pp. 690–703, http://dx.doi.org/10.1007/978-3-030-37453-2_55.

[25] C. Klinkmüller, A. Ponomarev, A.B. Tran, I. Weber, W.M.P. van der Aalst, Mining blockchain processes: Extracting process mining data from blockchain applications, in: Business Process Management: Blockchain and Central and Eastern Europe Forum, in: LNBIP, vol. 361, Springer, 2019, pp. 71–86, http://dx.doi.org/10.1007/978-3-030-30429-4_6.

[26] R. Hobeck, C. Klinkmüller, H.M.N.D. Bandara, I. Weber, W.M.P. van der Aalst, Process mining on blockchain data: A case study of augur, in: Business Process Management: International Conference, in: LNCS, vol. 12875, Springer, 2021, pp. 306–323, http://dx.doi.org/10.1007/978-3-030-85469-0_20.

[27] 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: Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track At Business Process Management, in: CEUR, vol. 2973, CEUR-WS.org, 2021, pp. 111–115.

[28] A. Koschmider, F. Duchmann, Extraction of meaningful events for process mining from blockchain, in: Blockchain and Robotic Process Automation, Springer, 2021, pp. 13–29, http://dx.doi.org/10.1007/978-3-030-81409-0_2.

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