A Data Extraction Methodology for Ethereum Smart Contracts

Flavio Corradini, Alessandro Marcelletti, Andrea Morichetta, Barbara Re
Computer Science Division
University of Camerino, Camerino, Italy
{name.surname}@unicam.it

***Abstract*—The broader adoption of blockchain for creating decentralised applications has raised interest in employing analysis techniques to support continuous improvement. Data extraction is crucial in this context, as it permits a better understanding of how applications behave. However, due to the variety of data sources (e.g., transactions and events) and the characterisation of the blockchain structure, several challenges arise in automatically extracting data. In particular, retrieving smart contract state changes remains unexplored despite its potential usage for discovering unexpected behaviour. For such reasons, this work proposes a methodology and a supporting tool for extracting data from smart contract executions and state changes. The obtained data is then offered in a way that can be easily converted to purpose-specific standards. The methodology was tested on the PancakeSwap Ethereum bridge smart contract.**

***Index Terms*—Blockchain, Process mining, Data extraction, Ethereum, Smart contracts, State changes.**

I. Introduction

The Ethereum blockchain is gathering interest thanks to its key features, including security, transparency, and decentralisation. This trend is also confirmed by the constant growth of blockchain-based applications that range from the exchange of digital assets to goods tracking [1]. In such a context, the extraction of data generated by the execution of blockchain-based applications is particularly useful to support their continuous improvement. Indeed, thanks to smart contracts, decentralised applications are executed directly in the blockchain, generating data that can be used for certified auditing and monitoring activities [2]–[7].

Among the different analysis techniques, process mining is an emerging solution for analysing blockchain applications exploiting data (i.e., logs) resulting from smart contract executions [8]. Process mining is a set of techniques that can be used to identify bottlenecks and deviations from the expected behaviour of the monitored processes [9], [10]. In particular, the process mining community is moving toward object-centric event data representing the backbone of managing and analysing complex process data [11]. The blockchain context can benefit from such an analysis but still lacks a suitable extraction methodology to process the complexity of data generated by blockchain-based applications. In this paper, we do not discuss any analysis, but we concentrate on data extraction as the most time and effort consuming task in an analysis (improvement) project, typically requiring more than 80% of resources [12].

Considering blockchain applications, the execution of a smart contract generates data that is stored in blocks (e.g., timestamp), transactions (e.g., sender, inputs, gas, and more), events and storage (i.e., the memory containing the smart contract state) [13]. Additional effort is also required to decode information that cannot be easily interpreted in its original form in the blockchain. Thus, the extraction activity have to deal with the heterogeneity of storage and decoding factors.

Moreover, catching the state changes of a contract permits a comprehensive understanding of the application and enables detailed analysis of the contract evolution over time. Differently from transaction and block, a state change does not generate a clear and accessible track, requiring a deep investigation of the low-level data structure [14], [15]. In Ethereum, each variable influencing the state of a smart contract is permanently stored and encoded in the storage memory based on a specific slot. In the case of simple variables, this slot is statically assigned, while for complex types (e.g., mappings and structs), this 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 Ethereum smart contracts including execution-related data and state changes**. To this aim, our methodology first captures the knowledge about the contract transactions and, for each of them, extracts the related state changes. This is possible by replicating transactions inside the Ethereum Virtual Machine (EVM) and obtaining the traces generated to reconstruct smart contract variables changes history. Usually, this leads to exploiting Ethereum archive nodes requiring a size of several TB of memory, depending on the client being used, and to define ad-hoc solutions requiring strong domain knowledge [14], [15]. Our methodology relies on a resource-efficient solution in terms of used technologies (i.e., massive data storage) and information. Indeed, our proposal permits the extraction of traces without the need to have an archival node or other kind of heavy data sources. To demonstrate the feasibility of the proposed solution, the methodology was implemented as a web application that makes the data extraction of a smart contract

accessible by taking the contract details from the user as input. We illustrate the benefits of our methodology using the PancakeSwap Ethereum smart contract, but it can be generally applied to any Ethereum smart contract.

The rest of the paper is organised as follows. Section II introduces information on the Ethereum blockchain's relevant characteristics. Section III describes the proposed methodology by focusing on the different steps and the adopted solutions. Section IV discusses the methodology in practice. Finally, Section V provides an overview of related works, and Section VI concludes the paper by pointing out future works.

II. Background

Ethereum is a decentralised, open-source blockchain with smart contract functionality [17]. 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 whose 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, which runs on the Ethereum Virtual Machine (EVM), similar to traditional programming languages. To write smart contracts, Ethereum provides the *Solidity*[1] programming language. 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.

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

III. The Extraction Methodology

This section describes the proposed data extraction methodology focusing on the challenges of extracting smart contract state changes. Figure 1 depicts the steps of the proposed methodology. 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 first retrieves the contract code and contract transactions. Then, the compilation and extraction of data related to contract states, events, transactions, and blocks is conducted. Finally, the results are inserted into a JSON log and provided to the user. Here, below, we analyse each step, providing more detailed information.

Get contract code. This step takes in input the *contract address* and the *contract name* from which to extract data and retrieve the *contract code* later used for further operations. To provide a fully automatised procedure, we assume that the contract code is verified and has a publicly available source code that is readable through the Etherscan API.

Get contract transactions. The scope of this step is to get all *contract transactions* from which data is extracted. Also, in this case, the input required by the user is limited to the previously inserted address. The extraction can be further refined by applying filters based on a block range.

Compile contract. Once the smart contract source code is obtained, we used the Solidity compiler to obtain 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 specified by Ethereum. It contains 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

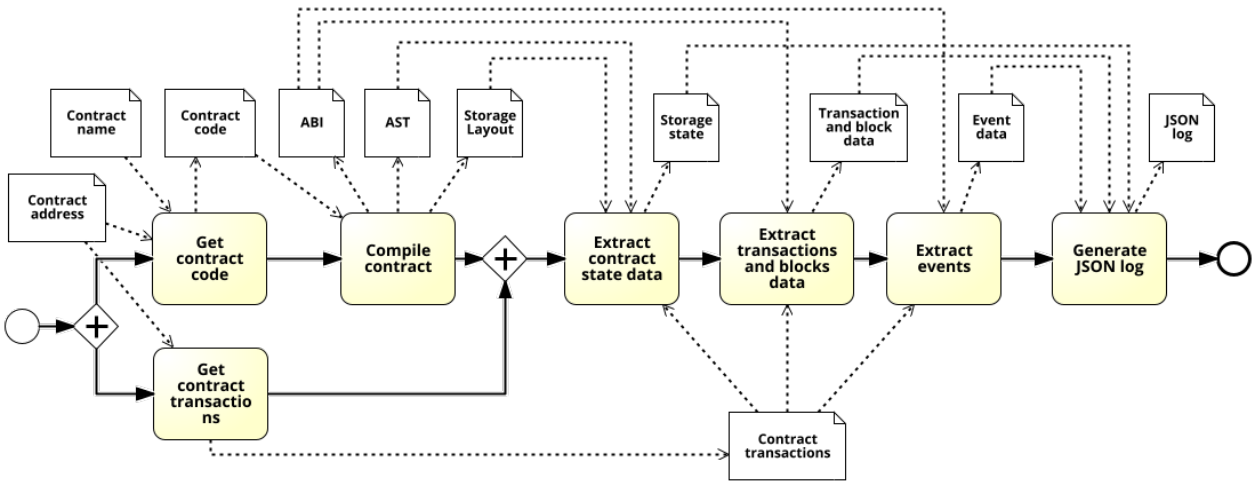

Fig. 1: Proposed data extraction methodology.

information extracted from transactions and events since they are stored in the blockchain in a hexadecimal format.

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. By analysing the generated AST, the methodology creates a mapping between the functions in the contract and the state variables they modify. This mapping is combined with the storage layout during the contract state extraction to recognise which state variables are updated in the executed functions.

The *Storage layout* interface represents how contract state variables are persistently stored in the storage memory. In this memory, the EVM stores the smart contract data using a key-value format. In simple state variables, the key represents the index of the memory slot, while for dynamic variables, the key is generated by hashing the index of the slot concatenated with a dynamic value (e.g., mapping key, array index). This poses a significant challenge in associating a specific storage key with a dynamic state variable. For this reason, in the case of simple variables, we use the storage layout to derive their memory location directly, while in the case of dynamic ones, we use this information as a starting point to compute the final location.

Extract contract state data. This step is the focal point of the proposed methodology, and it extracts the state variables updated after each contract execution. For this purpose, each transaction is replicated on a local environment with the state of the blockchain at the moment in which the transaction was originally executed. This returns the transaction trace containing the list of executed operations (i.e., opcodes) and the state of the EVM (i.e., memory locations). By analysing such operations and their inputs/outputs, it is possible to reconstruct the history of state variable changes and retrieve their location in the case of dynamic ones. As a result, the step produces the *storage state* that is later included in the final log. In the following, we describe the main standard opcodes existing in the EVM and how they are used in the proposed methodology. Indeed our solution relies on native functionalities, without the need for any customisation to the EVM.

STOP indicates that the execution of a certain smart contract is terminated. The resulting trace execution includes the final updated contract state, represented by the mapping between storage keys and their values. Notably, these keys are in hexadecimal format and represent the result of the simple conversion of the storage slot or the Keccak256 of the slot plus the dynamic values. This step matches each storage value by decoding the key to its related variable, thanks to information obtained from the storage layout. Indeed, in the case of simple variables, a conversion of the storage key returns the integer representing the variable location, permitting its mapping to the variable name contained in the storage layout and AST. However, in the case of dynamic variables, the keys cannot be directly decoded without additional information about their construction.

SSTORE is a simple opcode operation that saves state variable values inside the respective storage keys. This makes it possible to catch each new insertion or update in the storage. *SHA3* is the operation for the Keccak256 encryption of an input read from the memory. In particular, the SHA3 is also used to compute the storage key of dynamic state variables by hashing the default assigned storage slot and the dynamic value. The final hexadecimal value represents the storage key previously obtained through the STOP opcode in the contract state. By reading the sequence of SHA3 operations, we can derive the original storage slots used to create a certain key and, consequently, understand to which variables it is associated, thanks to the storage layout and the AST.

CALL, DELEGATECALL, STATICCALL are the codes referring to contract-to-contract executions. 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

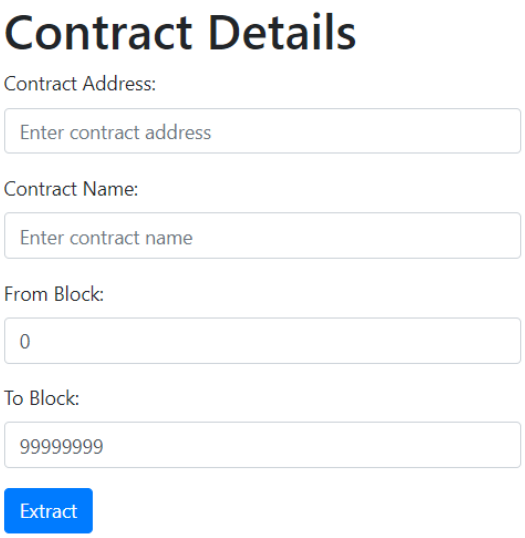

Fig. 2: Web application for extracting data.

for read-only operations without modifying 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.

Extract transactions and blocks data. Once the contract state changes are collected and decoded, the methodology continues to read information associated with transactions and blocks. For each of them, the methodology takes the name of the executed function from the corresponding log and its inputs, decoded thanks to ABI. Then, other attributes are read, such as hash, sender, timestamp, gas used, and more. This *transaction and block data* is saved for use in the last step.

Extract events. This last step concerns the retrieval of the events emitted by the smart contract functions and contained in the transaction log. Using the ABI, events of past transactions are captured together with the name and the value of the attributes. Also, in this case, the output *event data* is saved for the latest step.

Generate JSON log. The last step of the methodology creates the *JSON* log containing all the extracted data. This log is stored and provided to the user, who can use it for different analysis scopes.

IV. Extraction in Practice

To demonstrate the feasibility of the proposed methodology, in this section, we report a concrete example of extracting data from 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[2] representing the Cake token used for cross-chain capabilities.

The methodology was implemented as a web application based on Nodejs and is publicly available here. It exposes a user interface as illustrated in Fig 2, providing easy access to the methodology. Users can simply insert the contract address, name, and specify the range of blocks for a restricted extraction. The interaction with the blockchain relies on Web3js libraries[3] 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 API[4] is used. The source code compilation is done with Solc[5], and it requires a contract written in a Solidity with a minimum version of 0.5.13, as it introduced the feature generating the storage layout interface. To replicate transactions locally and obtain traces, the hardhat framework is used with the *debug_traceTransaction* functionality [6].

Example of data extraction. Listing 1 reports an entry from the output JSON log containing the data extracted from one transaction. The full log is available here. and it contains the first one hundred transactions of the contract.

The first JSON property contains the "sendFrom" *activity* (line 2), representing the function executed in the smart contract and extracted from the *inputData* field of the transaction. Once decoded, this field also returns the execution *timestamp* (line 3) and the eventual inputs. The latter ones are composed of the names of the variables *input names* (lines 4-6), their types *input types* (lines 7-9), and their values *input values* (lines 10-15). After function data, the log contains the *storage state* updates thanks to the transaction trace locally replicated (lines 16-22). Inside the JSON log, each state variable is an object having a *name* (lines 17, 20), a *type* (lines 18, 21), and a value (lines 19, 22) property. Currently, the decoding is not supported for nested dynamic variables, and the raw value is provided. In addition to state changes, the trace permits contract-to-contract invocations without relying on third-party services. This is visible in the *internal transactions* property (lines 23-27) containing a list of JSON objects for such transactions. Initially, each call is defined by its *type* (line 24), which can be CALL, DELEGATECALL, or STATICCALL, as in the reported example. Then, the address of the target contract is included in the *to* property (line 25). If the call also has some *inputs*, their value is taken (lines 26-27). This value is expressed in the encoded format, so additional information (e.g., ABI) about the call and the target contract is needed to decode it. Finally, the emitted *events* are reported (lines 28-34). Each event is represented by a JSON object, having a *name* (line 29) and the *values* (line 30), which are event-specific attributes and could differ inside each function.

V. Related Works

The employing of automatic methodologies for the extraction of blockchain data has experienced many proposals over the years. Particular interest was encountered in process

[2]https://bit.ly/3Og5gi4

[3]https://web3js.readthedocs.io/en/v1.2.11/getting-started.html

[4]https://docs.etherscan.io/api-endpoints/contracts

[5]https://docs.soliditylang.org/en/latest/using-the-compiler.html

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

```
{
"activity": "sendFrom",
"timestamp": "1680523403",
"inputNames": ["_from", "_dstChainId",
  "_toAddress", "_amount",
  "_minAmount", ...]
"inputTypes": ["address", "uint16",
  "bytes32",  "uint256",
  "uint256", ...],
"inputValues": [
  "0xA13bb...cA56E",
  102,
  "0x00000...ca56e",
  989898989898990000,
  989898980000000000, ...],
"storageState": [
  {"name": "failedMessages",
  "type": "mapping(..(..., bytes32)))",
  "value": "00000...c066b3fb96f01641"},
  {"name": "_balances",
  "type": "mapping(address, uint256)",
  "value": 449699370000000000}, ...],
"internalTransactions": [
  {"type": "STATICCALL",
  "to": "00000...bc225cd675",
  "inputs": ["9c729da10...8f7ea",
  "d6d4c898...00000"]}, ...],
"events": [
  {"name": "Transfer",
  "values": {
  "from": "0xA13bb...cA56E",
  "to": "0x00000...00000",
  "value": 9898989800000000000}}, ...]
}
```

Listing 1: Example of log entry for a single transaction.

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 [8]. A first attempt was proposed in [18] to discover business processes executed on the blockchain. In this framework, an XES log is derived from data related to blocks and transactions. Similarly, another early research described in [4] extracts transactions and generates XES logs for process discovery and conformance checking analysis. In [19], [20] instead, events emitted by Ethereum smart contracts are extracted and formatted according to XES for process discovery. The approach was then extended in [21], including the Hyperledger Fabric blockchain support. Fabric was also used in [22], 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 [13]. ACEL

Work	Contract type	Considered Data						Output format
		B	CC	CS	E	IT	T	
[18]	process based	✓	✗	✗	✗	✗	✓	XES
[4]	generic	✗	✗	✗	✗	✗	✓	XES
[19]	generic	✓	✗	✗	✓	✗	✗	XES
[20]	generic	✓	✗	✗	✓	✗	✗	XES
[21]	generic	✓	✗	✗	✓	✗	✗	XES
[22]	process based	✗	✗	✗	✓	✗	✗	CSV
[13]	generic	✓	✗	✗	✓	✗	✗	ACEL
[23]	generic	✓	✓	✗	✓	✓	✗	OCEL
proposed work	generic	✓	✗	✓	✓	✓	✓	JSON

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

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, [23] 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 I, and it considers the adopted (i) blockchain, the (ii) contract type, the (iii) extracted data, and the (v) 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 [21], [22]. As application types, some works assume smart contracts representing business processes instances [18], [22], 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 [23] 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 [23], and an extension was also provided in [13]. 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.

VI. Conclusions and Future Work

The continuous growth of blockchain is encouraging the creation of decentralised applications, relying on smart contracts as a form of disintermediated and trusted logic. Data produced by these applications during smart contract execution is considered transparent and immutable inside the blockchain, paving the way for innovative analysis techniques. Therefore, extracting and processing smart contract data assume a crucial role in enabling the usage of such data in different contexts like testing, monitoring, and process mining. This latter uses blockchain logs according to specific standards to enhance processes and identify irregularities. When applying this kind of analysis 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, some methodologies were proposed to automatically extract and convert blockchain data to particular standards, enabling the application of process mining techniques but without considering the smart contract state. For this reason, in this work, we present a data extraction methodology for dealing with the challenges of smart contract state extraction. For each generated transaction, the state of the contract and other execution-related information are extracted and inserted in a JSON log. In this way, it is possible to have a general-purpose representation that can be converted according to a standard depending on different needs. The feasibility of the methodology was tested on the Ethereum PancakeSwap token by extracting data and generating the corresponding JSON. As future improvements, we will support a larger amount of data related to smart contracts. Also, we plan to provide a converter that automatically formats the intermediate JSON to a desired standard.

Acknowledgment

Acknowledgement of the financial support of the project PNRR MUR project ECS_00000041-VITALITY.

References

[1] A. G. Gad, D. T. Mosa, L. Abualigah, and A. A. Abohany, "Emerging trends in blockchain technology and applications: A review and outlook," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 9, pp. 6719–6742, 2022.

[2] C. D. Ciccio, G. Meroni, and P. Plebani, "Business process monitoring on blockchains: Potentials and challenges," in *Enterprise, Business-Process and Information Systems Modeling*, vol. 387 of *Lecture Notes in Business Information Processing*, pp. 36–51, Springer, 2020.

[3] C. D. Ciccio, G. Meroni, and P. Plebani, "On the adoption of blockchain for business process monitoring," *Softw. Syst. Model.*, vol. 21, no. 3, pp. 915–937, 2022.

[4] F. Corradini, F. Marcantoni, A. Morichetta, A. Polini, B. Re, and M. Sampaolo, *Enabling Auditing of Smart Contracts Through Process Mining*, p. 467–480. Berlin, Heidelberg: Springer-Verlag, 2022.

[5] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, and F. Tiezzi, "Engineering trustable and auditable choreography-based systems using blockchain," *ACM Trans. Manag. Inf. Syst.*, vol. 13, no. 3, pp. 31:1–31:53, 2022.

[6] T. Cippitelli, A. Marcelletti, and A. Morichetta, "Chorssi: A bpmn-based execution framework for self-sovereign identity systems on blockchain," in *Business Process Management: Blockchain, Robotic Process Automation and Educators Forum*, vol. 491 of *Lecture Notes in Business Information Processing*, pp. 5–20, Springer, 2023.

[7] F. Donini, A. Marcelletti, A. Morichetta, and A. Polini, "Restchain: a blockchain-based mediator for REST interactions in service choreographies," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pp. 245–248, ACM, 2023.

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

[9] W. M. P. van der Aalst, "Process mining: A 360 degree overview," in *Process Mining Handbook* (W. M. P. van der Aalst and J. Carmona, eds.), vol. 448 of *Lecture Notes in Business Information Processing*, pp. 3–34, Springer, 2022.

[10] W. M. P. van der Aalst, *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[11] W. M. P. van der Aalst, "Object-centric process mining: Unraveling the fabric of real processes," *Mathematics*, vol. 11, no. 12, p. 2691, 2023.

[12] K. Diba, K. Batoulis, M. Weidlich, and M. Weske, "Extraction, correlation, and abstraction of event data for process mining," *WIREs Data Mining Knowl. Discov.*, vol. 10, no. 3, 2020.

[13] L. Moctar-M'Baba, N. Assy, M. Sellami, W. Gaaloul, and M. F. Nanne, "Process mining for artifact-centric blockchain applications," *Simul. Model. Pract. Theory*, vol. 127, p. 102779, 2023.

[14] K. Diba, K. Batoulis, M. Weidlich, and M. Weske, "Extraction, correlation, and abstraction of event data for process mining," *WIREs Data Mining Knowl. Discov.*, vol. 10, no. 3, 2020.

[15] J. D. Weerdt and M. T. Wynn, "Foundations of process event data," in *Process Mining Handbook* (W. M. P. van der Aalst and J. Carmona, eds.), vol. 448 of *Lecture Notes in Business Information Processing*, pp. 193–211, Springer, 2022.

[16] L. Moctar-M'Baba, M. Sellami, W. Gaaloul, and M. F. Nanne, "Blockchain logging for process mining: a systematic review," in *55th Hawaii International Conference on System Sciences*, pp. 1–10, ScholarSpace, 2022.

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

[18] R. Mühlberger, S. Bachhofner, C. D. Ciccio, L. García-Bañuelos, and O. López-Pintado, "Extracting event logs for process mining from data stored on the blockchain," in *Business Process Management Workshops*, vol. 362 of *Lecture Notes in Business Information Processing*, pp. 690–703, Springer, 2019.

[19] C. Klinkmüller, A. Ponomarev, A. B. Tran, I. Weber, and 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*, vol. 361 of *Lecture Notes in Business Information Processing*, pp. 71–86, Springer, 2019.

[20] R. Hobeck, C. Klinkmüller, H. M. N. D. Bandara, I. Weber, and W. M. P. van der Aalst, "Process mining on blockchain data: A case study of augur," in *Business Process Management - 19th International Conference*, vol. 12875 of *Lecture Notes in Computer Science*, pp. 306–323, Springer, 2021.

[21] P. Beck, H. Bockrath, T. Knoche, M. Digtiar, T. Petrich, D. Romanchenko, R. Hobeck, L. Pufahl, C. Klinkmüller, and 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*, vol. 2973 of *CEUR Workshop Proceedings*, pp. 111–115, CEUR-WS.org, 2021.

[22] A. Koschmider and F. Duchmann, "Extraction of meaningful events for process mining from blockchain," *Blockchain and Robotic Process Automation*, pp. 13–29, 2021.

[23] R. Hobeck and I. Weber, "Towards object-centric process mining for blockchain applications," in *Business Process Management: Blockchain, Robotic Process Automation and Educators Forum*, vol. 491 of *Lecture Notes in Business Information Processing*, pp. 51–65, Springer, 2023.