

Management of blockchain-based data provenance and addressing smart contract security issues

by

Shlomi Linoy

Master of Business Administration, College of Management Academic Studies, 2009
Bachelor of Science (Computer Science), College of Management Academic Studies,
2005

A Dissertation Submitted in Partial Fulfilment of the Requirements for the Degree
of

Doctor of Philosophy

In the Graduate Academic Unit of Computer Science

Supervisor(s): Suprio Ray, Ph.D., Computer Science
Natalia Stakhanova, Ph.D., Computer Science,
University of Saskatchewan

Examining Board: Rongxing Lu, Ph.D., Computer Science
Erik Scheme, Ph.D., Electrical & Computer Engineering
Martin Wielemaker, Ph.D., Business Administration

External Examiner: Jelena Mistic, Ph.D., Computer Science,
Toronto Metropolitan University

This dissertation is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

March, 2023

© Shlomi Linoy, 2023

Abstract

Blockchain gained massive popularity in recent years in industry as well as academia due to its specific properties that enable distrustful parties to mutually and pseudo-anonymously manage information through enforceable rules (in the form of smart contracts), and in a decentralized way. The eager adoption of the technology has led to converting and managing a considerable amount of funds in the form of cryptocurrencies. On the other hand, the ease of access to the public blockchain, in addition to the enforcement of pseudo-anonymous transactions, has incentivized its use in criminal activities, as well as numerous security related attacks. Consequently, groups of interest and authorities have been motivated to explore traceability and accountability of users to enable regulatory enforcement, and to prevent and mitigate security related issues.

In this work we present different approaches to address the above issues. To enable traceability and audit of smart contracts (abbreviated as contracts) and to analyze, detect, and mitigate contracts' security issues we present the EideticEther and EtherProv frameworks. The frameworks collect contracts' execution flow, including their accessed data, across time and in different granularities. Specifically, the EtherProv framework collects execution flow provenance at the control flow graph level, across all participating contracts. The collected provenance facilitates root-cause and forensic analysis, detection of security issues, and traceability, with the aid of provenance retrieval capabilities. Moreover, EtherProv enables mitigating

deployed contracts that exhibit undesired activities. To help deanonymize pseudo-anonymous addresses we present an approach that utilizes stylometry techniques to extract unique features of Ethereum contracts' code that can represent the coding style of the contracts' developers. We explore the feasibility of using these features to attribute contracts' code to their deployer's address, and consequently, affiliate addresses that were used to deploy contracts written by the same developer. In order to enable the described approaches, efficient management and retrieval of historical data is required. However, current blockchain indexes enable to query a single key and its latest value. Our proposed AMVSL blockchain index enables efficient authenticated historical data management and their retrieval over a large range of keys and their current or historical values. To enable rigorous regulatory enforcement auditors require frequent access to multiple blockchains. However, due to the rapid increase of blockchain data volume and their inefficient querying capabilities of a large amount of data, maintaining local blockchain nodes for querying purposes can prove inefficient. To this end, we propose a system that enables remote auditing of blockchain data, providing efficient and richer queries while supporting private information retrieval by utilizing cryptography techniques over semi-trusted servers to protect the auditors' identities, queries and their results. To handle large data volumes the system employs a scalable distributed processing solution for big data.

Dedication

To my beloved family. Thank you for helping me reach my full potential and achieve my goals.

Acknowledgements

I am deeply grateful to my supervisors, Dr. Suprio Ray and Dr. Natalia Stakhanova, for their unwavering guidance and exceptional support throughout my academic journey. Their invaluable insights and encouragement have made this experience not only fruitful but also enjoyable.

Additionally, I would like to express my gratitude to my thesis advisory committee members, namely Dr. Erik Scheme, Dr. Rongxing Lu, Dr. Martin Wielemaker, and Dr. Jelena Misic, for taking the time to carefully read and provide valuable feedback on my thesis. Their thoughtful suggestions and constructive criticisms have significantly enhanced the overall quality of my work.

I extend my sincere appreciation to Dr. Erik Scheme and Dr. Rongxing Lu for their readily extended support, valuable discussions, and feedback.

Furthermore, I would like to thank Dr. Hassan Mahdikhani and Dr. Alina Matyukhina for their assistance and input in the collaborative publications and research.

Moreover, I am grateful to Dr. Paul Cook for the enjoyable discussions on how to critically review papers at the beginning of my studies.

Last but not least, I would like to express my thanks to all the UNB students and staff that I had the privilege to meet.

I am grateful for the opportunity to have met each one of you during this meaningful time.

Table of Contents

Abstract	ii
Dedication	iv
Acknowledgments	v
Table of Contents	vi
List of Tables	xv
List of Figures	xvii
Abbreviations	xviii
1 Introduction	1
1.1 Problem statement	3
1.1.1 Comprehensive historical provenance in blockchains with smart contract support	4
1.1.2 Smart contracts security analysis	5
1.1.3 Pseudo-anonymous address affiliation	7
1.1.4 Historical provenance data management	8
1.1.5 Remotely accessing blockchain data securely and efficiently . .	8
1.2 Contributions	9
1.2.1 Addressing security issues through provenance analysis in blockchains with smart contract support	10

1.2.2	Efficient blockchain provenance management	12
2	Background	15
2.1	Blockchain	15
2.1.1	Ethereum blockchain	17
2.2	Ethereum blockchain execution environment	21
2.2.1	Access control	22
2.2.2	Execution resources	22
2.2.3	Code patching	22
2.2.4	Historical data analysis	22
2.3	Provenance	23
3	Related Work	25
3.1	Provenance	25
3.2	Addressing security issues through provenance analysis in blockchains with smart contract support	27
3.2.1	Static analysis	27
3.2.2	Dynamic analysis	29
3.2.3	Transaction-based security analysis	30
3.2.4	Blockchain addresses clustering	31
3.2.5	De-anonymization of blockchain addresses using out-of-network information	32
3.2.6	Authorship attribution	33
3.2.7	Ethereum smart contract scams	34
3.3	Efficient blockchain provenance management	35
3.3.1	Authenticated data structures	35
3.3.2	Querying blockchain data	36
3.3.3	Blockchain data retrieval	36

4	Generic blockchain-based provenance-aware system	39
4.1	Addressing security issues through provenance analysis in blockchains with smart contract support	41
4.2	Efficient blockchain provenance management	42
5	EideticEther: towards eidetic blockchain systems with enhanced provenance	45
5.1	Smart contract execution flow provenance	46
5.1.1	Motivating retail sector example	47
5.1.2	Implementation details	49
5.2	The EideticEther framework	51
5.2.1	Collecting provenance data	51
5.2.2	Querying provenance data	52
5.3	Health insurance sector example	55
5.3.1	Implementation details	55
5.3.2	Querying provenance data	57
5.4	Comparison to related work	58
6	EtherProv: provenance-aware detection, analysis, and mitigation of Ethereum smart contract security issues	59
6.1	The EtherProv framework	64
6.1.1	Smart contract execution provenance collection	67
6.1.1.1	Generating extended CFG	67
6.1.1.2	CFG efficient path profiling	68
6.1.1.3	Instrumenting Solidity source code	73
6.1.1.4	Dynamic data extraction	74
6.1.2	Security analysis	75
6.1.3	Tracking & Mitigation	76

6.2	Comparison to related work	77
6.3	Enabling dynamic analysis with higher degrees of soundness and completeness through Solidity source code static analysis data	81
6.4	Blockchain smart contract security issues	82
6.5	Security Evaluation	84
6.5.1	Implementation	84
6.5.2	Detecting known smart contract vulnerabilities	85
6.5.2.1	Liquid Ether	85
6.5.2.2	Re-Entrancy	86
6.5.2.3	Restricted Writes	88
6.5.2.4	Detecting violations across contracts	88
6.5.3	Analyzing new security threats in deployed contracts	90
6.5.4	Mitigating security threats in deployed contracts	95
6.6	Performance evaluation	97
6.7	Complexity evaluation	99

7 De-anonymizing Ethereum Blockchain Smart Contracts through Code Attribution 102

7.1	Difference between smart contract code and “traditional” non-blockchain code	106
7.2	Smart contract attribution	108
7.2.1	Extracting Ethereum smart contracts	110
7.2.2	Feature selection step	110
7.2.3	Heuristic refinement step	112
7.3	Evaluation	115
7.3.1	Data	115
7.3.2	Evaluation results	116
7.3.3	Feature selection	117

7.3.4	Heuristic refinement results	121
7.3.5	Attribution results	122
7.4	Exploring distinctly contributing features	124
7.5	Real-world scams	130
7.5.1	Attributing Ponzi scheme smart contracts and examining their distinctly contributing features	130
7.5.2	Attributing real-world scammers	131
7.6	Comparison to related work	133

8 Authenticated Multi-Version Index for Blockchain-based Range

	Queries on Historical Data	139
8.1	Problem definition	143
8.2	Proposed approach	145
8.2.1	Index structure overview	146
8.2.2	Versioning bucket data structure overview	147
8.2.3	Upsertion algorithm	149
8.2.4	Deletion algorithm	155
8.2.5	Index commit	157
8.2.6	Index authentication process	157
8.2.7	Query processing algorithms	159
8.2.8	Efficiently querying all keys across a range of versions	164
8.2.9	Implementation of SVRK, MVRK, and MVAK queries	168
8.3	Comparison to related work	169
8.4	Evaluation	170
8.4.1	Systems evaluated	170
8.4.2	Dataset	171
8.4.3	Experimental setting	171
8.4.4	Experiments	172

8.4.4.1	Writes throughput on dataset “Varied number of keys”	172
8.4.4.2	Query latency on dataset “Varied number of versions”	174
8.4.4.3	Query latency on dataset “Varied number of keys”	174
8.4.4.4	Query latency on 1M keys, 10 versions, and varied keys/versions search percentage	175
8.4.5	Complexity Analysis	177
8.4.5.1	AMVSL	177
8.4.5.2	MBT and MPT (without multi-version or range search support)	178
8.4.5.3	MBT and MPT with multi-version and range search support	178

9 Scalable Privacy-Preserving Query Processing Over Ethereum

Blockchain		180
9.1	The proposed system	182
9.1.1	Main components	183
9.1.2	Assumptions:	184
9.1.3	Privacy preserving query processing	185
9.2	System Model	185
9.3	Comparison to related work	193
9.4	Evaluation	193
9.4.1	Fetching missing data from Geth clients	195
9.4.2	Fetching data from HDFS repository	195
9.4.3	Steps following server data fetching	196

10 Conclusion and future work

10.1	Conclusion	197
10.2	Future work	201

10.3 Closing thoughts	203
Bibliography	219
A	220
A.1 Eidetic blockchain frameworks with Enhanced Provenance	220
A.1.1 Retail example implementation details	220
A.1.2 Health insurance example implementation details	221
B	223
B.1 The EtherProv framework	223
C	229
C.1 Caliskan source code feature set	229
C.1.1 Lexical features	229
C.1.2 Layout features	231
C.1.3 Syntactic features	232
Vita	

List of Tables

5.1	Partial contracts description	50
5.2	Clients data	50
5.3	Suppliers initial data	50
5.4	Client order state changes across session - contract call parameters . .	53
5.5	Client order state changes across session - API function parameters .	53
5.6	Client order state changes across session - suppliers database row after contract call	53
5.7	Client order state changes across session - contract call parameters . .	54
5.8	Client order state changes across session - API function parameters .	54
5.9	Client order state changes across session - client orders database row after contract call	54
5.10	Health insurance - contracts' API description	57
5.11	Insurance scenario - contract call parameters	58
5.12	Insurance scenario - API function parameters and deductibles after contract call	58
6.1	contract_call_changed_states query results	92
6.2	Instrumented contracts statistics	97
6.3	Comparing EtherProv to related work	101
7.1	Feature sets per code type	111
7.2	contracts' dataset statistics	116
7.3	Inspected classifiers and feature selectors	117

7.4	RF classification accuracy per feature selector	119
7.5	Statistics of distinctly contributing features for source code unigram, bytecode unigram, and Caliskan feature sets encompassing all authors (<i>selected4contracts</i> dataset was used with a <i>Heuristic 5</i> refinement) .	128
7.6	Top 20 of 42 most distinctly contributing source code unigrams en- compassing all authors (a DT feature selector was used on the <i>se-</i> <i>lected4contracts</i> dataset, which was refined with a <i>Heuristic 5</i> refine- ment)	128
7.7	Top 20 of 49 most distinctly contributing source code Caliskan fea- tures encompassing all authors (a DT feature selector was used on the <i>selected4contracts</i> dataset, which was refined with a <i>Heuristic 5</i> refinement)	129
7.8	Caliskan features legend	129
7.9	Ponzi scheme dataset statistics	130
7.10	Statistics of distinctly contributing source code unigrams encompass- ing all authors and the Ponzi author specifically (a DT feature selec- tor was used on the combination of the <i>selected4contracts</i> and <i>Ponzi</i> datasets refined with <i>Heuristic 7</i>)	130
7.11	Top 20 of 39 most distinctly contributing source code unigrams of the Ponzi author (a DT feature selector was used on the combination of the <i>selected4contracts</i> and <i>Ponzi</i> datasets refined with <i>Heuristic 7</i>) .	131
7.12	Scammers dataset statistics	133
7.13	Feature set per heuristic refinement statistics	137
7.14	RF classification results per feature selector, feature set, and heuristic	138
8.1	Multi-version and single-version range queries	144
8.2	Synthetic uniform random dataset details	171
8.3	Complexity analysis summary	179

8.4	Complexity parameters description	179
9.1	Query examples	186
9.2	Query Q1 Fetch/Processing extracted queries example	186
9.3	Block numbers and ranges used	193
9.4	Single and multiple nodes configuration	194
A.1	Retail - contracts' API description	220
A.2	Health insurance - contracts' API description	222

List of Figures

1.1	Contributions and the cross-cutting concerns they address and supplement	14
2.1	Block and transaction data structures in the Ethereum blockchain	16
2.2	High-level flow of an Ethereum contract's EVM deployment and its corresponding source code retrieval	20
4.1	Mapping between the blockchain-based provenance-aware system aspects and our proposed approaches	40
5.1	Execution flow of a full order request session	49
5.2	Scenario implementation overview	49
5.3	A session's contract calls' graph	52
5.4	Suppliers contract calls Shipping_request on Shipping contract	55
5.5	Insurance scenario execution flow	56
5.6	Scenario implementation overview	56

6.1	EtherProv Framework overview	65
6.2	EtherProv detailed static and dynamic analysis data extraction flow	66
6.3	Extended CFG	68
6.4	Efficient path profiling - CFG instrumentation and paths extraction .	71
6.5	Efficient path profiling - illustration of four path types	71
6.6	Solidity source code CFG instrumentation	73
6.7	EtherProv security analysis flow overview	75
6.8	EtherProv tracking & mitigation flow overview	77
6.9	Sampled results of decoded path from EtherProv query	93
6.10	Scalability analysis of Listing 6.9 query	95
6.11	Instrumented contracts' average gas overhead CDF	96
6.12	Instrumented contracts	97
7.1	Account address affiliation using smart contract code attribution . . .	103
7.2	High-level flow of Ethereum smart contracts authorship attribution .	109
7.3	Preliminary classification accuracy using RF and SVM classifiers using cumulative top features that are extracted by RF, DT, MI feature selectors	118
7.4	Finding cumulative classification accuracy of minimal top ranked fea- tures using a RF feature selector and classifier	120
7.5	Finding cumulative classification accuracy of minimal top ranked fea- tures using a DT feature selector with a RF classifier	120
7.6	An example DT that was produced by the feature selector	124
8.1	Versioned relational schema for a table T	142
8.2	AMVSL - skip list structure	146
8.3	AMVSL - bucket structure (partition capacity of 2)	147
8.4	An example of AMVSL index	149

8.5	AMVSL index scenario after inserting key 150	151
8.6	AMVSL - versions to active keys data structure	164
8.7	Insert throughput evaluation on “Varied number of keys” dataset . .	172
8.8	Query latency evaluation on dataset “Varied number of versions” . .	173
8.9	Query latency evaluation on dataset “Varied number of keys”	175
8.10	Query latency evaluation on 1M keys, 10 versions, and varied keys/ver- sions search percentage	176
9.1	Flow of user query execution: <i>SELECT MAX(value) FROM transactions</i> <i>WHERE block_number BETWEEN 100 AND 2000</i>	183
9.2	System Components and Model	186
9.3	Evaluation of fetched blocks data from (a) Geth clients and (b) HDFS	194
9.4	Performance breakdown following data fetch	196
A.1	Database schemes for all departments	221
A.2	Health insurance scenario - database schema details	221

Abbreviations

- ADS** Authenticated Data Structures. 35
- AMVSL** Authenticated Multi Version Skip List. 12
- API** Application Programming Interface. 36
- AST** Abstract Syntax Tree. 34
- CDF** Cumulative Distribution Function. xvi, 96
- CFG** Control Flow Graph. 28
- Contract** Smart Contract. 1
- DAG** Directed Acyclic Graph. 69
- DAO** Decentralized Autonomous Organization. 27
- DDos** Distributed Denial of Service. 27
- DFS** Depth First Search. 99
- DT** Decision Tree. 117
- ECF** Effectively Callback-Free. 30
- EOA** External Owned Account. 18

EVM Ethereum Virtual Machine. 1

HDFS Hadoop Distributed File System. 181

IoT Internet of Things. 2

LOC Lines Of Code. 116

MBT Merkle Bucket Tree. 35

MI Mutual Information. 117

MPT Merkle Patricia Trie. 35

MST Maximum Spanning Tree. 100

MVAK Multi-Version on All Keys. 142

MVRK Multi-Version on Ranged Keys. 142

P2P Peer To Peer. 32

POW Proof Of Work. 16

PT Physical Therapy. 55

RF Random Forest. 111

SMT Satisfiability Modulo Theories. 28

SSA Static Single Assignment. 67

SVM Support Vector Machine. 111

SVRK Single Version on Ranged Keys. 142

TSE Traditional Software Environment. 21

USD United States Dollar. 2

UUID Universal Unique Identifier. 189

Chapter 1

Introduction

The blockchain technology enables different pseudo-anonymous parties who do not trust each other to share information without the need for a central authority, using a robust consensus protocol. In 2008 Satoshi Nakamoto introduced the seminal bitcoin blockchain [104] that enabled to pseudo-anonymously exchange funds in the form of a bitcoin, which is regarded as the first crypto-currency, without the need for a central authority such as a bank. In 2014 Wood et al. proposed the Ethereum blockchain [137], which enabled, in addition to the exchange of funds in the form of Ether, the use of smart contracts. A *smart contract* (abbreviated as *Contract*) is a piece of computer code that resides on the blockchain and enables users to create their own arbitrary enforceable rules for ownership and state transition functions, in a decentralized way. Each contract maintains its own consistent state on the blockchain, in the form of key-value pairs, which are modified according to the executed contract's flow. Contracts are written in a high-level language such as Solidity and are compiled into bytecode, e.g., Ethereum Virtual Machine (EVM) bytecode in the Ethereum platform. The contract bytecode is then deployed to the blockchain to enable its use. Users execute a contract by creating a transaction that contains the address of the contract, the function to be called, and the required function pa-

rameters. A contract can transfer Ether to user addresses (i.e., Externally Owned Addresses) and to other contracts, create new contracts, and call functions in other contracts.

Due to the exciting potential of the blockchain technology to empower systems with decentralized, distributed, tamper-resistant, and fault-tolerant capabilities, blockchain systems exploded in popularity in recent years and captured the attention of industry as well as academia. As a result, huge amounts of funds are being traded in the blockchain ecosystem with an overall market capitalization of over \$913bn USD as of October 12, 2022 [17]. Ethereum [137], which is the first platform to support contracts, remains the most popular with the second largest market capitalization of over \$157bn USD as of October 12, 2022 [17]. There are over 50 blockchain platforms available today [5]. Many of them enable the creation and automated execution of contracts. Among them are Ethereum [137], Tezos [75], EOS [3], Cardano [2], and Hyperledger Fabric [7].

Blockchain systems with contract support have been rapidly adopted by the industry to manage valuable assets with practical applications in numerous sectors such as healthcare [15], government [18], IoT [16], insurance policies [72], securities trading [19], law enforcement [16], and identity management [96]. For example, contracts can facilitate claim processing speed up, reduce operating costs in law enforcement sectors, and enable online decentralized secure voting [140]. The health care sector can benefit by adopting blockchain solutions to establish a standardized system where patient records are stored and analyzed without revealing private information, where the embedded crypto-currency can be integrated to handle financial aspects of the system [79, 102]. Multinational retail corporations face recurring supply chain management issues, such as the one related to Walmart's recent romaine lettuce E. coli outbreak in North America [71]. The exact source and extent of the contaminated lettuce could not be determined from the supply chain system, which resulted

in huge losses to all parties involved. In order to reduce such issues, companies are adopting solutions based on blockchain technologies at a high rate [128].

Since blockchain can serve as a trustworthy decentralized storage solution for handling structured data, it has inspired research in the database and systems community [31, 53, 110, 139]. Multiple research approaches make use of *data provenance*. Data provenance can be defined as metadata that describes various details of the data creation such as what additional data were used to create the described data, why they were used to create the described data, how they were used, and what is *their* data provenance. The blockchain innately tracks data provenance at the transaction level, e.g., in Ethereum each transaction records information such as transaction initiator address, account/contract destination address, and the amount of Ether sent. This information provides details on how the stored data were created and changed, and how they flow across transactions and addresses, across the entire history of the blockchain. While the blockchain technology is still being explored alongside its potential domains, different aspects of the technology are being studied.

1.1 Problem statement

The blockchain enables to pseudo-anonymously manage funds in the form of cryptocurrencies in a decentralized way. While these properties can enable truly free commerce, at the same time they can be abused. Specifically, the pseudo-anonymity property ensures that although all transactions' information is publicly and transparently available on the blockchain, the identities behind the addresses are unknown without out-of-network information. This, in turn can promote:

1. Illicit activities. Not knowing the real identities of the parties that are involved in a transaction can circumvent regulation of activities. Indeed, huge amounts of supervised funds have been reallocated to the blockchain in a short amount

of time.

2. Security attacks. The substantial amount of funds managed in a public blockchain and the ease of access to a public blockchain using a pseudo-anonymous addresses can incentivize attacks.

These challenges have motivated special interest groups and authorities to explore efficient ways to:

1. Provide comprehensive and detailed traceability of blockchain users and their activities in order to enable regulatory enforcement and accountability.
2. Enable analysis, prevention, and mitigation of security related issues and their resulting monetary losses, and to ensure the safe adoption of the technology.

In order to address these requirements various cross-cutting aspects of the blockchain technology need to be addressed, as we discuss in the following sections.

1.1.1 Comprehensive historical provenance in blockchains with smart contract support

A comprehensive and detailed historical provenance in blockchains with smart contract support should encompass all blockchain data as well as detailed contract execution information, across time. As discussed above, the blockchain innately tracks historical data provenance at the transaction level, which contains the sender's and receiver's pseudo-anonymous addresses, the amount of funds transferred, the time of the block inclusion and its comprising transactions; and, if the transaction is used to issue a contract call, the executed contract's address, its called function name, and its parameters. The *current digest* of the blockchain state, which is a single hash that is computed from the hashes of all current state values, is also maintained as part of the transaction, as opposed to the state *values*. The state values are stored

in a secondary key-value store, where a subset of these state values are changed after each transaction execution. Hence, the historical state values are not retained by the blockchain. As in any computer program, a contract's execution traceability, or execution provenance, is required to understand how the contract changed the blockchain's state, from what initial state, and the resulting state. Managing the comprehensive historical provenance of both data and execution provenance enables to maximize traceability and to facilitate a comprehensive understanding of blockchain users' activities, across time. We note that contract execution provenance is not retained by the blockchain. A comprehensive and detailed historical provenance, with the aid of provenance query capabilities, can help facilitate traceability and accountability, root-cause and forensic analysis, as well as analysis, mitigation, and prevention of security issues. In existing studies [98, 118] the contract execution provenance is either used ad hoc and then discarded or only partially stored in a limited fashion. Hence, there is a need to enable efficient extraction and management of historical data as well as execution provenance to enable the above capabilities.

1.1.2 Smart contracts security analysis

Since blockchain technology increasingly entail significant monetary value to their users, the benefits of this technology have been overshadowed by numerous security concerns [48, 92]. In recent years, a number of reports exposed contracts' vulnerabilities and exploits, which mainly stem from the immaturity of the field, and consequently, a lack of knowledge and tools for automated analysis and verification of contracts. Current approaches for automated analysis and verification of contracts make use of either static analysis or dynamic analysis methods. In dynamic analysis the program *is run* for a subset of inputs and their outputs are analyzed, as a result the analysis is confined to extracting precise information from executed paths. Since covering all possible paths may lead to the path explosion problem most approaches

utilize static analysis [38, 66, 78, 83, 130, 133], which analyzes an over-approximation of the program execution as a whole *without running it*. Static analysis approaches are mostly used to detect *known* contract vulnerabilities while dynamic analysis approaches are mostly used to analyze *unaddressed* security issues in contracts' executed flows. Current dynamic analysis approaches [131, 143] exclusively operate on traces of executed EVM bytecode. Enabling dynamic analysis of Solidity source code control flow graph (CFG) paths can provide a more efficient, accurate, and timely dynamic analysis. Such an analysis will be akin to following an execution flow on the Solidity source code, which has several benefits over operating on EVM bytecode traces, e.g., any user with an understanding of the Solidity language can perform the dynamic analysis, which will not be confined to security experts alone; further, the Solidity CFG's executed path contains composite structures such as arrays or mappings, which allow to efficiently and precisely determine the memory access flow. In contrast, EVM traces contain discrete memory addresses, which are cumbersome to consolidate to their corresponding logical data structure. Hence, a dynamic analysis of the Solidity source code CFG can facilitate a timely, precise, and efficient analysis. A comprehensive analysis approach that leverages both static and dynamic analysis of Solidity source code constructs can enable detecting known contract vulnerabilities before running the contract and efficiently analyzing unaddressed security issues in the contracts' executed flows. Further, by managing the collected static and dynamic analysis data in a unified data schema, dynamic analysis based insights of unaddressed security issues can be efficiently represented as patterns that can be used by the static analysis to detect future similar security issues. Currently, no approach supports an integrated static and dynamic analysis capabilities of contracts, or dynamic analysis of Solidity source code CFG.

1.1.3 Pseudo-anonymous address affiliation

Blockchain security analysis and mitigation issues are further aggravated due to pseudo-anonymous addresses. Blockchain users are identified by addresses (public keys), which cannot be easily linked back to them without out-of-network information. This provided pseudo-anonymity is amplified when a user generates a new address for each transaction, which cannot be easily linked to one another or to the user. Since all transaction history is visible to all the users of public blockchains, pseudo-anonymity is essential not only to protect the privacy of users' transactions but also to hinder address affiliation that can be used to discriminate against addresses that are linked with undesired activities. For this reason, in the case an address was involved in a malicious behavior or a security attack, even if the address was linked to the attacker, considerable and often manual effort is required to associate the attacker with other related addresses. Therefore, automatic affiliation of addresses is essential to determine the full scope of a security attack spanning all affected addresses, which in turn can provide a more efficient analysis, mitigation, and resolution of the issue. Further, in order to enforce accountability, address affiliation can help in de-anonymizing addresses when additional out-of-network information is available. Current address affiliation approaches rely on over-approximating user behavior assumptions. For example, addresses are assumed to be affiliated with the same user if they share a transaction [44], transfer remaining funds to other addresses [127] (i.e., change addresses in bitcoin), or used as inputs in the same transaction [99, 113]. However, due to the premise of the blockchain pseudo-anonymity it is challenging to validate the affiliation accuracy of these approaches or determine their effectiveness. As a result, additional address affiliation approaches are required to provide different insights for possible cross reference, which can increase the confidence of law enforcement analysis results. Specifically, approaches with some degree of verifiability.

1.1.4 Historical provenance data management

As discussed above, to enable security and traceability related capabilities, comprehensive historical provenance is required that includes historical data and execution flow. At the same time, to adhere to regulatory requirements, this historical data should be authenticated and verifiable. Current blockchain systems offer little to no support for managing and querying historical data. In addition, current blockchain indexes, e.g., MPT [137] (used in Ethereum) and MBT [51] (used in Hyperledger Fabric), support storing and managing only current data. Although a full blockchain archive node can be used to maintain historical data of all executed transactions off-chain, this is not scalable due to the consistently and rapidly expanding data volume. Moreover, such offline historical data lack authentication and tamper-resistance assurance as their verification is not part of the blockchain's consensus protocol. In addition to adherence to regulations, verifiable and authenticated data can be consumed by contracts on run-time and increase their analysis capacity. To support these capabilities, there is an increasing need to enable efficient on-chain management and querying of authenticated and verifiable historical data to be consumed by smart contracts to enable richer analysis capabilities as well as to provide efficient offline analysis without requiring to duplicate the already considerable amount of data.

1.1.5 Remotely accessing blockchain data securely and efficiently

Due to the rapid expansion of blockchain systems' data, a blockchain node, and specifically, a blockchain archive node, requires a considerable amount of storage. Further, current blockchain solutions store the blockchain data in an append-only and immutable fashion, which provides limited and inefficient querying capabilities.

To address these limitations, current approaches [55, 55–59, 62, 63, 65, 93] rely on fetching the blockchain data to a secondary database, optimizing the stored data using indexes, and exposing different APIs to query the data. However these approaches contain the following limitations: (1) a set of predefined queries limits the overall querying capabilities, (2) all blockchain data need to be fetched from a single blockchain node using a standard protocol, which is inefficient, and stored in the database before serving any query request, and (3) the main domain is restricted to public blockchain data, which is not suited for securely querying sensitive private or semi-private blockchain data. As a result, querying private or semi-private blockchain data efficiently and securely can prove challenging for external entities, e.g., auditors that require frequent access to such, possibly multiple, resources. This challenge can be addressed by enabling remote access to existing blockchain nodes to query the required data securely. Further, to enable rigorous audit procedures, the privacy of the auditors and the queries should be preserved, as well as the retrieved results. Consequently, to address these issues there is a need for an approach that enables querying blockchains remotely, efficiently, and in a privacy-preserving manner.

1.2 Contributions

In this work, we propose a generic blockchain-based provenance-aware system that enables to efficiently track and manage current and historical contracts’ execution data to address blockchain traceability (regulatory) and security issues. To support this vision, we design a set of approaches to ensure data security and traceability, and to enable an efficient data provenance management and query capabilities. In the following, we discuss our approaches and note the cross-cutting concerns they address and supplement, which is summarized in Fig. 1.1.

1.2.1 Addressing security issues through provenance analysis in blockchains with smart contract support

Contribution 1: Exploring smart contracts’ comprehensive traceability through dynamic provenance In order to explore the analysis capabilities resulting from a comprehensive and detailed historical provenance that includes contracts’ execution data, in chapter 5 we propose EideticEther, a framework that efficiently extracts and manages historical information of contract calls, their parameters, and the blockchain state before and after a contract call. We explore the capabilities of EideticEther using scenarios from the retail and health insurance sectors. In the retail sector, we explore how the collected historical provenance can be used to query a contract call’s execution flow across time to facilitate more complete understanding of the contracts’ execution environment, enable more detailed traceability capabilities, and quality and maintenance management. In the health insurance sector, we explore how the collected historical provenance can be used to perform root-cause analysis across time. The proposed approach addresses the cross-cutting concern raised in 1.1.1 and 1.1.2.

Contribution 2: Addressing smart contracts’ security issues through static and dynamic provenance As discussed, EideticEther enables tracing contract execution provenance at the contract function call graph level, which can detail the sequence of contract function calls, and for each function call provides the initial state value and the resulting value at the end of the function call. However, to enable more efficient and detailed root-cause and forensic analysis, and specifically security analysis, the details of *how* a contract function changed its input state is required. Current approaches that analyze security issues in smart contracts make use of either static analysis (on Solidity source code or on EVM bytecode) or dynamic analysis (on the EVM bytecode alone). Where each approach contains its own strengths and

limitations. Hence, an approach that supports both static and dynamic analysis can benefit from both worlds. To this end, in chapter 6 we propose EtherProv, a comprehensive provenance-aware framework that leverages static provenance (collected offline) and dynamic provenance (collected online) synergy to enable the analysis, detection, and mitigation of security issues in Ethereum contracts, as well as richer traceability capabilities. EtherProv leverages Solidity source code static and dynamic provenance through contract bytecode instrumentation. The collected data are transformed into a unified, high-level representation, which can be queried using concise and descriptive Datalog queries. Within the provenance framework, EtherProv enables analyzing contracts’ execution flow over time, detecting vulnerabilities and analyzing attacks within a single contract execution flow and across multiple interacting contracts, and mitigating compromised *deployed* contracts. Our evaluation shows that EtherProv can efficiently identify vulnerable contracts with an average contract instrumentation gas overhead of 18.9%. The proposed approach addresses the cross-cutting concerns raised in 1.1.1 and 1.1.2.

Contribution 3: Addressing pseudo-anonymous address affiliation In chapter 7 we propose an approach to effectively deduce affiliation provenance of addresses that are used to deploy contracts created by the same contract developer. In our approach, we leverage stylometry techniques, which extract a contract author’s unique stylometry characteristics from their Ethereum source code and bytecode. A similar approach has been widely used in the software field [67, 70]. To validate our approach, we prepare a real-world dataset that contains all addresses that were used to deploy contracts and their related Solidity code and bytecode. We use a conservative approach and assume that each user is identified by a single account address, which can be used to deploy multiple contracts. Our evaluation on this dataset validated that our approach can accurately classify contract code to their deployers’ addresses using stylometry techniques, which in turn can affiliate multiple deployers’ addresses

that were used to deploy contracts' code with similar characteristics. We show that even a small number of representative features lead to sufficiently high accuracy in attributing contracts' code to their deployer's address. We further validate our approach on real-world scammers' data and Ponzi scheme contracts. Additionally, we provide an algorithm to extract distinctly contributing features per an entire dataset or per specific authors. We use this algorithm to extract and explore such features in our prepared real-world dataset and in the Ponzi scheme dataset. The proposed approach addresses the cross-cutting concern raised in 1.1.3 and supplements approaches that address the cross-cutting concerns raised in 1.1.1 and 1.1.2.

1.2.2 Efficient blockchain provenance management

Contribution 4: Addressing efficient blockchain provenance management

As discussed, authenticated and verifiable historical data is essential to ensure regulation adherence as well as security analysis capabilities. To address these challenges, in chapter 8 we propose an authenticated index structure called Authenticated Multi-Version Skip List (AMVSL), which is designed to take part in the blockchain's consensus protocol. The index enables efficient historical blockchain data management and rich and efficient set of query features, which retrieve authenticated data over a large range of keys and their current or historical values. We further present three range queries: SVRK, MVRK and MVAK, which offer querying over a range of keys and a range of versions. Our experimental evaluation demonstrates that AMVSL can efficiently support these queries and can achieve better performance over existing authenticated data structures. The proposed approach addresses the cross-cutting concern raised in 1.1.4 and supplements approaches that address the cross-cutting concerns raised in 1.1.1, 1.1.2, and 1.1.5.

Contribution 5: Addressing remote access to blockchain data in a secure and efficient manner

To enable querying blockchains remotely, efficiently, and in a privacy-preserving manner, in chapter 9 we propose a system that is designed to operate “on top” of an existing blockchain, which enables performing efficient and richer queries over blockchain data while supporting private information retrieval by utilizing cryptography techniques over semi-trusted servers to protect the auditors’ identities, queries and their results. To handle the current and rapidly increasing blockchain data volume, the system employs a scalable distributed processing solution for big data. The proposed approach addresses the cross-cutting concern raised in 1.1.5 and supplements approaches that address the cross-cutting concerns raised in 1.1.1, 1.1.2, and 1.1.4.

Before proceeding to the proposed approaches, in Chapter 2 we provide additional background on blockchain and its relevant components and procedures, in Chapter 3 we discuss the related work, in chapter 4 we discuss how the different approaches contribute to the vision of a generic blockchain-based provenance-aware system and how they supplement one another to provide richer system wide capabilities, and in Chapter 10 we provide our conclusion and discuss future work.

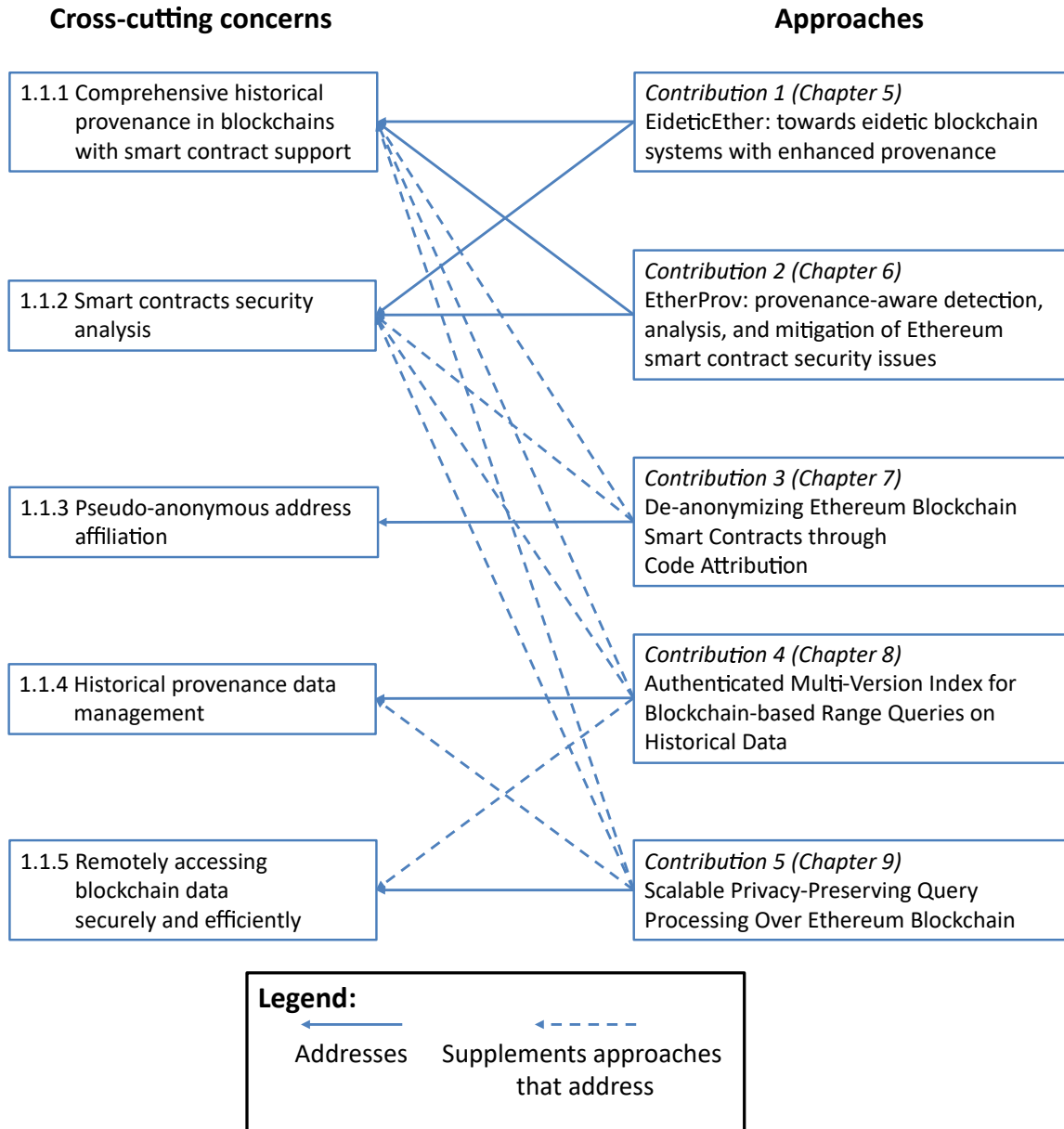


Figure 1.1: Contributions and the cross-cutting concerns they address and supplement

Chapter 2

Background

In this chapter we provide some background that introduces the concepts used throughout this work. We provide further details about the general blockchain and its comprising components, i.e., block, transaction, full node, consensus protocol, miners, and pseudo-anonymity. We continue to introduce the Ethereum blockchain and its specific blockchain details, i.e., smart contract, gas, account types, Ether, event, transaction, Ethereum data, data queries, Ethereum full node, contract deployment overview, and retrieving the contract’s corresponding Solidity code. We next compare the Ethereum blockchain execution environment to “traditional” execution environments in terms of access control, execution resources, code patching, and historical data analysis. We conclude with our definition of provenance, in general, and specifically for computer programs.

2.1 Blockchain

Blockchain uses cryptographic pointers to link immutable blocks and transactions in a chronologically-ordered linked chain. The blocks are linked to their predecessors by references provided in the block-header. The blocks and their contained transactions can be resolved individually by a cryptographic hash. Fig. 2.1 illustrates a simplified

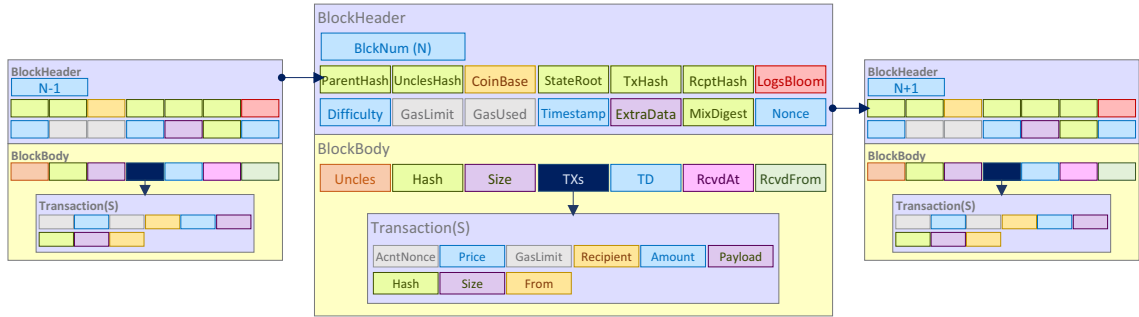


Figure 2.1: Block and transaction data structures in the Ethereum blockchain

architecture of the Ethereum blockchain’s block and transaction structures and their relations. We next introduce the general blockchain comprising components.

Block Each block points to its predecessor block by a cryptographic hash pointer, which is stored in the block-header. The cryptographic hash pointer incorporates a hash of the combined data and meta data of its previous block, which ensures the immutability and integrity of the information in the previous blocks. Each block contains multiple transactions.

Transaction A message sent between two blockchain addresses, which changes the blockchain state.

Full node Contains its own replica of the blockchain data, including the contracts’ current states. It is also referred to as a shared ledger.

Consensus protocol Blockchain technologies enable different parties who do not trust each other to share information using a robust consensus protocol, which eliminates the need for a central authority. The most prominent consensus protocol in use is Proof-of-Work (POW) [104], which requires a mining node to solve a time and resource consuming computational puzzle to confirm a block of transactions. This synchronization process is also called mining.

Miners Full nodes that create a new block through the consensus protocol are called miners.

Pseudo-anonymity In public blockchains, the ledger, which contains all historical activities that were made between any two addresses, is publicly available. However, blockchain addresses hide the real identities of their owners (users), which cannot be obtained without out-of-network information. This provided pseudo-anonymity makes it challenging to establish which transactions were issued to/from the same user.

2.1.1 Ethereum blockchain

Bitcoin [104] is the most popular blockchain platform, which enables trading bitcoin crypto-currency. Ethereum [137] is the second most popular blockchain platform. While bitcoin is mainly a crypto-currency, Ethereum enables, in addition to crypto-currency, the creation and running of contracts in a decentralized way. Fig. 2.1 illustrates a simplified architecture of the Ethereum blockchain's block and transaction structures, and their relations. We next discuss the components of the Ethereum blockchain that are relevant to our work.

Smart contract A smart contract (abbreviated as *contract*) is an enforceable agreement, which is realized as a computer code that enables users to create their own arbitrary rules for ownership and state transition functions. The contract is written in a high-level language such as Solidity [20] and is compiled into Ethereum Virtual Machine (EVM) bytecode. The EVM bytecode is deployed to the Ethereum blockchain and is provided with an address, which can be used to interact with the contract. The invocation of a contract contains the contract's address, the function name to call and its parameter values. A contract can also be called from other contracts. The contract maintains its own state, which can be changed according to

the program flow.

Gas In order to ensure finite and concise execution, each contract operation consumes some amount of *gas*, which is a measurement unit used to calculate the amount of funds to be paid for the operation. If the gas runs out before the contract finishes its execution, it is terminated.

Account In Ethereum there are two types of accounts (addresses): External Owned Account (EOA) - an address that is derived from a public key that is generated and owned by an external entity (or user), and a Contract Account - an address that is generated from the combination of the deployer's address and the number of transactions that were sent from the deployer's address until the deployment of the contract.

Ether Ethereum's crypto-currency is called *Ether*, which can be managed by an EOA or contract.

Event Contracts contain an immutable and tamper-resistant logging mechanism. A contract can interact with this logging mechanism using *events*. When a contract invokes an event, its data are emitted (logged) to the transaction's events, i.e., each transaction is associated with its own events.

Transaction Transactions contain the following data: the sender's address (*from* address), the receiver address (*to* address), and the amount of Ether to transfer to the *to* address. Each transaction requires a transaction fee to incentivise its execution by the miners. There are three types of Ethereum transactions that are used to (1) transfer Ether between two EOAs, (2) deploy a contract's EVM bytecode and receive its deployment address (in this case the *to* address is empty), and (3) initiate a contract call (the *to* address should contain the contract's address. In addition, the called function encoding and its parameters should be provided). Transactions can

only be initiated by an EOA. In the context of contracts this means that any contract call is the result of an EOA contract invocation, i.e., the EOA called a contract directly or the EOA called a contract, which called another contract. Subsequently, only the first contract call, which was initiated by the EOA, is stored in the transaction. The resulting contract calls can be reproduced by replaying the transaction with the initial contract call.

Ethereum data The EVM execution makes use of a stack machine with a depth of 1024 items, where each item is a 256-bit word, and is used to store the operands' computation results in each instruction. The EVM uses transient and persistent memory models. The transient memory is comprised of a word-addressed byte array that does not persist between transactions. The transient memory includes the stack as well as memory addresses that are used to store intermediate computation results. The persistent memory (referred to as *storage* or *blockchain state*) is a key-value store, which is stored in an internal database and not in a transaction or a block. It is used to store data between transactions, e.g., Ether amounts and contracts' states. Each transaction can modify the storage state to a new state. Since previous states are not maintained, no change history is recorded, i.e., each transaction sees only the current snapshot of the storage state.

Data queries Each block or transaction is associated with a unique hash and number. Indexes are used to enable efficient queries on their values, i.e., index for block number, index for block hash, index for a transaction hash, and index for a transaction number within a specific block.

Ethereum full node We refer to an Ethereum full node as an Ethereum node. Each Ethereum node stores its own replica of the Ethereum blockchain data by downloading it from other Ethereum nodes. Each downloaded block is validated by

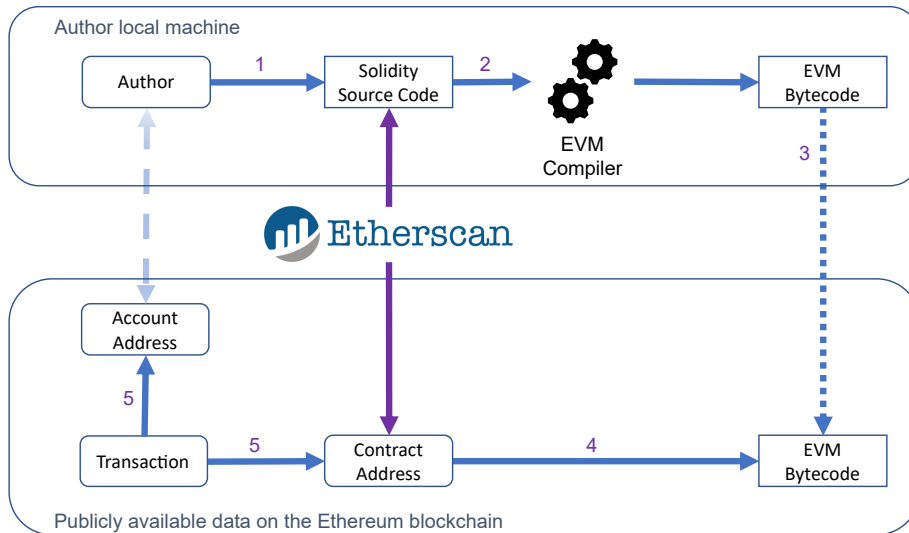


Figure 2.2: High-level flow of an Ethereum contract’s EVM deployment and its corresponding source code retrieval

recomputing its hash and comparing it to the downloaded block’s hash. The hash computation involves replaying the contained transactions and their data. When a transaction contains a contract call, the contract call is replayed by running the contract bytecode through an EVM execution engine using the transaction’s data and the current storage parameters’ snapshot, which may change at the end of the transaction.

Ethereum supports multiple client implementations across a range of different platforms. Go-Ethereum (Geth) and Parity are the leading Ethereum nodes that have been developed for different operating systems. They store the data in LevelDB [90] and RocksDB [114] respectively, which are key/value stores.

Contract deployment overview In order to execute a contract in the Ethereum blockchain, the contract’s EVM bytecode needs to be deployed to the blockchain. Fig. 2.2 shows the high-level flow of the process that results in this deployment.

(1) The author of the contract writes the contract’s code in a high-level Solidity language. (2) The Solidity source code is compiled to EVM bytecode. (3) The author initiates a transaction to deploy the EVM bytecode by providing their account

address and the EVM bytecode payload. (4) A contract address is assigned to the successfully deployed contract. (5) An additional transaction is created that links the newly created contract's address to the author's account address.

Retrieving the contract's corresponding Solidity code Since only the contract's bytecode is retained in the blockchain in this process, there is no way to retrieve the contract's original Solidity code from the blockchain alone. To enable the extraction of the deployed contract's Solidity source code, third-party Ethereum blockchain explorer platforms such as `etherscan.io` [23] can be utilized.

`etherscan.io` provides a range of capabilities that include the retrieval of contracts' source codes in specific cases. To achieve this, the Solidity source code is uploaded to `etherscan.io` (usually by the author). The source code is then compiled into bytecode and is compared with the deployed contract's bytecode. If the two are equal, the contract's source code is considered verified. Fig. 2.2 shows how using `etherscan.io` can provide the pairing of the Solidity source code to the corresponding contract address in the blockchain (purple arrows).

While the Solidity source code can be linked to the contract address in this way, the figure also illustrates the preserved pseudo-anonymity of the author who cannot be directly linked to their account address (light blue dashed arrow), and consequently, to the deployed contract's address or its provided Solidity source code. Using third-party Ethereum blockchain explorer platforms to publicly provide and pair the Solidity source code to its deployed bytecode is optional. However, to promote transparency and trust, contract authors are encouraged to do so.

2.2 Ethereum blockchain execution environment

The Ethereum blockchain execution environment differs from a Traditional Software Environment (TSE) in several aspects. We next discuss these differences.

2.2.1 Access control

In TSE the code is deployed to an environment that is owned and moderated by a specific entity, which can dictate the access control to the environment. In Ethereum, the environment can be publicly accessed, which includes all transactions' data and contract storage values.

2.2.2 Execution resources

In TSE, code execution is usually limited only by available allocated hardware resources. In Ethereum, each contract execution is limited by a predefined allocated amount of gas.

2.2.3 Code patching

In TSE, when a defect or a security issue is detected, the environment's code is updated with the fixed code. In Ethereum, deployed contracts' code are immutable and cannot be modified.

2.2.4 Historical data analysis

In order to analyze a historical contract call execution within a historical transaction, the Ethereum node requires the historical transaction's data (available on-chain) and the historical snapshot data state that was available before the historical transaction was executed (not available on-chain). The relevant historical snapshot data state needs to be provided. One way to reconstruct the historical snapshot data state involves replaying all preceding transactions sequentially, starting with the first transaction in the genesis block (first block in the blockchain). At the end of the last preceding transaction, the relevant contracts and Ether storage states are available and can be used as an input to run the required historical transaction.

The same constraints apply to TSE; however, unlike in blockchain systems, access to “genesis” states may not be feasible since the persistence of such data is not required in TSE.

2.3 Provenance

The Oxford English Dictionary defines provenance as “the source or origin of an object; its history and pedigree; a record of the ultimate derivation and passage of an item through its various owners.”. However, the definition of provenance changes according to the field, e.g., the Oxford Dictionary of Art and Artists defines *art* provenance as “The record of the ownership of a movable work of art. An unbroken provenance accounts for the whereabouts of a work from the time of its creation to the present day, and the nearer a work’s pedigree approaches this ideal, the more secure its attribution is likely to be.” In contrast to the general provenance definition, the later definition pertains specifically to a sub category of art (“movable work of art”) and provides the application of the pedigree information (assurance of the work’s attribution).

The category of computational tasks is broad and its applications are varied. Oliveira et al. [106] detailed the general application of a computational task’s provenance in the context of scientific experiments as a way to explain and understand the computed results. The results’ explanation is enabled by examining the history and pedigree of the data that were used in the computation of the results, which contributes to more insightful understanding. Additional provenance applications in this context include verifying the experiments were conducted correctly, and in some cases, enable their reproducibility.

The input to computational tasks is usually composed of computed data from other tasks. In order to understand these computed input data, the details of how each

input data was computed can provide more insightful understanding of the whole computation process, especially when the data volume is large. As a result, current research on data provenance extended the general provenance definition to record, in addition to the history and pedigree of the data, *how* the history and pedigree of the data were used to obtain the resulting data [47]. Since data are usually comprised of multiple parts, each part can be annotated with this extended provenance. Provenance annotation can be applied to processes as well as data. Such provenance can include how the process was devised, its motivation, its purpose, what other processes it uses, etc. Collecting additional relevant provenance can contribute to a more detailed explanation and understanding.

As we can see, the amount of provenance information can be very large. This creates challenges relating to what provenance should be extracted, can be extracted, how to extract it, and how to efficiently manage and retrieve it. Addressing these challenges is dependent on the domain and the specific requirements. Practical solutions may include a collection of different levels of provenance information abstraction/granularity to reduce the overall volume while still providing a sufficient approximation of the information.

Chapter 3

Related Work

In this chapter the related work is presented in detail. We divide the discussion into twelve parts, i.e., provenance, blockchain security attacks, static analysis, dynamic analysis, transaction-based security analysis, blockchain addresses clustering, de-anonymization of blockchain addresses using out-of-network information, authorship attribution, Ethereum contract scams, authenticated data structures, querying blockchain data, and blockchain data retrieval.

3.1 Provenance

Provenance, herein, refers to the origin and change history of data. It has been studied in different contexts, including databases, software engineering, scientific workflows, and programming languages among others.

Data provenance has been extensively studied in the database research community. In data provenance, each row in the output of a single query (which is possibly composed of sub queries) is annotated with the input tuples that derived it. Cheney et al. [47] introduced three types of data provenance for a specific output tuple in a query result: *Why-provenance* - the set of minimal input tuples that contributed to an output tuple; *How-provenance* - specifies how an output tuple was generated from

the minimal input tuples; and *Where-provenance* - maps the specific output tuple's fields to the input tuples' fields. Glavic et al. [74] present a system that supports all three types of data provenance by using query rewrites to annotate the output tuples with the corresponding data provenance.

Whereas data provenance deals with data content, *workflow provenance* looks at the data transformation process that is modeled by a predefined dataflow or control flow. Miao et al. [103] proposed a system that collects workflow provenance in a collaborative workflow environment. The provenance is collected for the workflow's components, configuration, inputs, and outputs, and enables analyzing the components' call graph. In the context of blockchain, each transaction can be regarded as a workflow component with its own configuration, inputs, and outputs, for which we can extract a workflow provenance. Information about the blockchain's global state changes withing each transaction can be regarded as data provenance.

Due to its potential, interest in blockchain-based data provenance has been steadily growing. Researchers explored using blockchain as a tamper-resistant, fault-tolerant, distributive, and decentralized database for storing the data provenance information itself. Liang et al. [94] presented a system that uses blockchain to store encrypted provenance information of cloud storage operations. Ramachandran et al. [112] explored using blockchain to store provenance information of sensitive medical research. Recently, Ruan et al. [118] proposed a system that enables blockchain contracts to efficiently access historical contract states, which can increase the contract's computation possibilities.

3.2 Addressing security issues through provenance analysis in blockchains with smart contract support

As the size of Ethereum blockchain grows considerably, so do the security concerns. Mauro Conti et al. [48] surveyed attacks in the bitcoin blockchain, which include double spending, bribery attacks, brute force, transaction malleability, 50% hash power, selfish mining, DDos attacks, and routing attacks. Xiaoqi Li et al. [92] provided a survey of attacks in Ethereum, which included the use of criminal contracts, exploitation of contracts' security vulnerabilities, and the DAO attack.

In recent years, a number of tools were developed to verify the security properties of contracts. These approaches can be broadly classified into static and dynamic analysis based approaches. Another distinction is made among the existing approaches in terms of whether they can perform transaction-based security analysis.

3.2.1 Static analysis

Static analysis inspects a program's code without the need to run it. A number of security analysis approaches rely on static analysis of the program's code. Since the program code may be composed of multiple paths that can quickly grow exponentially, analyzing all possible execution paths can be resource-consuming. To address this issue, a common approach is to broadly approximate all conceivable execution paths, which retains a higher completeness (identifying all potential issues in the code). However, this may come at the cost of reduced soundness (ensuring that each issue was correctly classified). The majority of the current blockchain security verification approaches leverage static analysis of Solidity source code or bytecode, which aims to discover the presence or absence of potential code security vulnerabilities.

Oyente [97] and Mythril [21] disassemble the contract’s EVM bytecode (static provenance), extract its control flow graph (CFG), determine all reachable paths by utilizing symbolic execution (static analysis), i.e., mathematical analysis of different program paths, and checks them for vulnerabilities. Symbolic execution can provide high soundness (high precision), however, since it is known to suffer from a path explosion problem, it does not provide a high degree of completeness (results in low coverage). Oyente [97] was one of the earlier tools for contract security verification where the authors analyzed three types of vulnerabilities. The follow-up studies have reported Oyente’s limited coverage (20.2% code coverage on Parity wallet [35]) and inability to accurately discover vulnerabilities [133].

Solc-Verify [78] leverages a formal verification approach. Using contract code specifications annotations, Solc-Verify confirms a contract’s properties using satisfiability modulo theories (SMT) solvers (as part of its static analysis). While the previous approaches work on EVM bytecode, Solc-Verify operates on the contract’s high-level Solidity source code.

Vandal [38] and Securify [133] disassemble the contract’s EVM bytecode, translate it to an intermediate language, extract its CFG, and translate the CFG bytecode constructs to semantic facts, which are stored in Datalog. Known security issues are represented as patterns that are used to query the Datalog intensional database for pattern violations. These patterns work on an abstracted representation of the contract CFG that is sufficient in most cases. Since these approaches over-approximate the reachability of all instructions, they provide a high degree of completeness but at the same time they can suffer from lower degree of soundness, which can lead to higher false positives, compared to the approaches using symbolic execution.

Slither [66] converts the Solidity source code CFG into an intermediate representation (IR). Similar to Vandal and Securify, it uses vulnerability patterns on an abstracted representation of the contract’s CFG. Different from these approaches, the vulnera-

bility patterns are hard coded and are composed of Solidity constructs, which make use of the accurate Solidity source code CFG, as opposed to the less accurate CFG that is extracted from the bytecode in the approaches taken by Vandal and Securify. With the prevalence of static analysis tools, there were some efforts to evaluate their effectiveness. Ghaleb et al. [73] showed that compared to others, Slither [66] provided the lowest false negative rate.

3.2.2 Dynamic analysis

In dynamic analysis, a subset of inputs is provided. The program is run for each input, and the outputs of each executed path are collected and analyzed. Dynamic analysis does not focus on the program as a whole, instead it operates only on executed paths according to the provided inputs. Completeness is not usually guaranteed, since covering all possible paths would be demanding in terms of computation and space consumption. Conversely, dynamic analysis helps ensure a higher degree of soundness, i.e., per provided input, the related path can be accurately analyzed. The dynamic analysis based approaches utilize run-time information for vulnerability analysis. These approaches generally leverage instrumentation to collect run-time metrics such as execution time, instruction count, and gas consumption. Compared to static analysis, dynamic analysis has been less widely adopted for security analysis of contracts.

In existing studies the contract's execution flow is extracted and analyzed using extensive instrumentation of an Ethereum node's EVM execution code [45, 46, 98]. For example, SODA modifies a full Ethereum node with extensive instrumentation. The instrumentation enables the collection of fine grained data including blocks, transactions, and contracts' data. Per contract execution, all read and written state values and their locations are collected. When collecting a written state's new value, its old value is preserved. The collected data are stored in a queue, which is read

asynchronously by different security detection applications. The applications use stack-based patterns to detect attacks. A similar instrumentation approach was also adopted in EVM*, which enables monitoring a contract function’s execution in real time by instrumenting the full node’s EVM execution environment. It also uses stack-based patterns to detect attacks. EVM* is one of the few systems offering a mitigation strategy for deployed contracts. The modified EVM is complemented with monitoring and interrupting mechanisms, which collect relevant opcodes, and determine if a transaction should be terminated or allowed to execute. However, its reliance on opcode level evaluation is resource consuming and is limited to vulnerabilities and bugs contained within a single function.

An additional distinction is made among existing approaches, in terms of whether they can perform transaction-based security analysis, which we discuss next.

3.2.3 Transaction-based security analysis

Transaction-based security analysis makes use of dynamic and static analysis, across blockchain transactions and time. It involves replaying historical transactions, using their provided inputs (dynamic analysis) and collecting the execution traces and data, and storing them into storage. Following, static analysis is performed on these traces to detect vulnerabilities or attacks in contracts within a single transaction trace or across transaction traces.

The latest transaction-centric approaches include Sereum [115], ECFChecker [77], TXSpector [143], and HORUS [131]. Sereum and ECFChecker focus on the the detection of the Re-Entrancy attack. Sereum enables detecting the Re-Entrancy attack using dynamic taint tracking. The Re-Entrancy attacks that can be detected are cross function Re-Entrancy, delegated Re-Entrancy (across contracts), and create-based Re-Entrancy (on contract creation). ECFChecker dynamically analyzes a contract’s execution to determine if it is an Effectively Callback-Free (ECF)

object. A contract’s execution trace that is deemed to be non-ECF is determined to exhibit the Re-Entrancy attack. TXSpector and HORUS provide an analysis framework that enables analyzing contracts’ dynamic execution trace using logic-driven program analysis. This is achieved by replaying historical transactions and extracting the transaction’s data and control flow dependencies from the EVM bytecode execution trace that are transformed into Datalog logic relations. The logic relations are then queried to analyze and detect different attack and vulnerability patterns. TXSpector, in addition enables a transaction execution’s forensic analysis.

A large-scale analysis of contracts for six well-known security vulnerabilities was conducted by Perez and Livshits [109]. They discovered that only 2% of vulnerable contracts are exploitable.

3.2.4 Blockchain addresses clustering

As blockchain user addresses are pseudo-anonymous it is challenging to find which user addresses were used by, or are associated to, the same user. In the following we discuss approaches that try to address this challenge. Reid et al. [113] analyzed the bitcoin network with the use of two abstractions: “transaction network” and “user network.” The “transaction network” shows the flow of bitcoins from one transaction to the next over time, where each input edge of one transaction node is the output edge of the previous transaction. On the other hand, the “user network” shows the flow of bitcoins from one user (payer) to another user (payee); each user is represented by a collection of their bitcoin addresses. Since a user can have multiple unconnected bitcoin addresses, there is no accurate process to determine which user is connected to which bitcoin address. The authors used the assumption that if addresses are used as inputs in the same transaction they are considered to belong to the same user. Meiklejohn et al. [99] use the same assumption on input addresses but correlates their validity with the number of times they are used together as inputs

in all transactions. Spagnuolo et al. [127] proposed to use a “change address,” which is the address used to send any funds that remain at the end of a transaction, as a method to cluster bitcoin addresses to the same user. Chan et al. [44] explore the feasibility of affiliating Ethereum addresses using transaction graph analytics where transaction data are transferred into a graph database and tags are collected from sources such as `etherscan.io`. Addresses are considered to be affiliated if they share a transaction. Norvil et al. [105] explored unsupervised clustering techniques based on the ssdeep hash similarity [86] of Ethereum contracts’ bytecode. To find the context of a specific cluster, the authors analyzed the Solidity source code for each contract in the cluster and extracted the most frequent tokens.

3.2.5 De-anonymization of blockchain addresses using out-of-network information

Since blockchain user addresses are derived from the users’ public keys, they are randomly generated. As a result, extrapolating the identity of an address’s user requires out-of-network information. In the following we present approaches that try to address this. Santamaria et al. [119] showed that publicly available metadata, which are associated with the bitcoin network, can be used to obtain additional information regarding a bitcoin address, e.g., in forums, users provide their bitcoin address in a posted question or as part of their message signature, which associates the user’s forum identity with their bitcoin address. Meiklejohn et al. [99] used bitcoin addresses from verifiable sources, e.g., goods vendors or exchanges, to follow the transactions and trends related to these bitcoin addresses. Since bitcoin operates on a peer to peer (P2P) network, the transaction’s issuer information can be obtained from the network infrastructure underlying the peer nodes, combined with information on the nodes that participate in the transaction relay. Koshy et al. [87] showed that anomalous transaction relays can help in deducing the connection be-

tween the bitcoin owner and their IP address. Kaminsky [84] presented a method of de-anonymization that connects the transaction’s input bitcoin address to the IP address of the first relayer. This method can be actualized if the attacker can connect to all nodes in the bitcoin network. However, the anomalous transaction behavior approach used by Koshy et al. [87] enables a better de-anonymization approach than that of the first relayer approach, although anomalous transaction behavior is much less frequent than non-anomalous. Biryukov et al. [36] discussed a de-anonymization method that uses “entry” nodes (all the peer nodes that the bitcoin client is connected to) where, if an attacker is connected to an “entry” node, the IP address can be forwarded to them.

While the majority of research has been conducted on bitcoin, it could be applicable to Ethereum as well. Klusman et al. [85] explored how the approaches proposed by Biryukov et al. [36] and Spagnuolo et al. [127] (discussed above) can be applied to Ethereum, with some changes to the Ethereum network.

3.2.6 Authorship attribution

In the field of authorship attribution, the characteristics of an author’s writings are inferred from the documents they produce. The underlying assumption is that each author can be distinguished from others through unique writing characteristics. The approach that is widely used in the social science field for attribution of literary texts to their author facilitates stylometry techniques. A similar approach is taken in the software field, which extracts a programmer’s unique stylometry characteristics from their programs’ code. State-of-the-art methods in source code authorship attribution rely on low-level information such as word or character n-grams. Such features have been widely used [67, 70] as they are able to capture stylistic information. At the binary level, the corresponding byte-level n-grams have also been successfully explored [68, 69]. More complex features, which require additional parsing of code,

were also explored, although less frequently due to the overhead and complexity. There have been several attempts to utilize syntactic (structural) features for author attribution tasks. Most notable among these works is the study by Caliskan et al. [43] that utilized syntactic features derived from abstract syntax trees (ASTs) and unigrams term frequency for attribution of code. In the work by Watson’s et al. [135] the authors focused on de-anonymizing C++ programmers who have contributed to a Github repository. The authors assumed that the programmers were the ones who wrote the code if their Git repository contained a commit statement that produced the specified function. For features selection the authors used three different feature extraction methods: character-level - provide insight into the function’s lexical and stylistic properties; token-level - provide information about the function’s syntax and semantics; and AST-level - capture information about the function’s structure and behavior. Subsequently, these features were utilized to train a random-forest classifier on a dataset comprised of 37 repositories with between 3 and 37 authors each (with total of 346 authors) with at least 50 complete functions throughout the commit history. The trained model was able to correctly determine authorship of a given function with an F1 measurement of 75%. For more information on various authorship attribution methods and challenges we refer the interested reader to a survey by Vaibhavi et al. [81].

3.2.7 Ethereum smart contract scams

Despite a rising number of abuses in blockchain, research on this topic remains limited. One of the recent studies offered an insight on a “smart” financial fraud pyramid scheme, called a Ponzi scheme, executed with contracts. Bartoletti et al. [35] analyzed Ethereum contracts involved in a Ponzi scheme estimating the investment in such contracts over a period of six months in 2018 to be ~630K USD. The authors offer a detailed analysis of fraudulent contracts and their classification according to

redistribution of payouts.

3.3 Efficient blockchain provenance management

3.3.1 Authenticated data structures

Authenticated Data Structures (ADS) were studied in various environments where the data server may not be entirely secure and can provide tampered results; hence the main motivation behind using ADS is to enable access to immutable data in a tamper evident manner. Recently, several ADS-based index structures have been proposed that are inspired by the Merkle tree [100]. A Merkle tree is a tree of hashes that is used to authenticate a list of items. Each list item's cryptographic hash (abbreviated as hash) represents a leaf of the Merkle tree. A non-leaf node's hash is computed from the hashes of its children, and the root node's hash can be used to verify the entire Merkle tree. Each list item can be verified by comparing its hash value to the hashes of its predecessor nodes on the path to the root.

Li et al. [91] proposed an ADS, in the context of outsourced databases that is based on a B^+ -Tree, called EMB-tree, where each node contains an additional hash value that is computed from its children's hash values. The state-of-the art blockchain systems manage their states in external ADS. Merkle Patricia Trie (MPT) [137] is a radix tree with cryptographic authentication. Each key defines a path in the MPT and is split into hex characters, called nibbles. Similar to the Merkle tree, the cryptographic hashes of all paths are combined at the root node to provide the verification of the entire MPT. Merkle Bucket Tree (MBT) [51] is a Merkle tree built on top of a hash table, where each of its entries is called a bucket. Within each bucket the entries are arranged in a sorted order. The bottom level of the MBT maintains the cryptographic hashes, which are computed from the contents of the hash table buckets, while the internal nodes are formed by calculating the cryptographic hashes

of their intermediate children.

3.3.2 Querying blockchain data

Blockchain systems share many goals with traditional database systems, particularly the fundamental problem of transaction management. This has led researchers to attempt to develop database-like capabilities on blockchain systems. Several projects focused on improving transaction performance, which include [121] and [142]. In advanced search and query features, blockchains still lag behind databases, as research in relational databases goes back several decades, whereas blockchain systems are relatively new. SEBDB [145] is a blockchain database that enables querying capabilities on top of a blockchain. FalconDB [108], a recently proposed system, supports historical query features. However, it implements these by utilizing external systems, including MySQL for data storage and querying, IntegriDB [144] for authentication, and Tendermint [39] as the underlying blockchain.

3.3.3 Blockchain data retrieval

There are some technical limitations to accessing multiple blocks and transactions via the Ethereum node. Ethereum defines a JSON-RPC stateless protocol [61] to access Ethereum blockchain data from an Ethereum node. web3.js [60] is used by the client to access the blockchain using the JSON-RPC protocol. Specifically, the JSON-RPC protocol defines the methods `eth_getBlockByNumber`, `eth_getBlockByHash` and `eth_getTransactionByHash`, to retrieve data of a specific block or transaction respectively. The protocol does not support an efficient retrieval of multiple block/transactions in one method call. Due to these protocol limitations, fetching blocks and transactions has been impeded by Ethereum nodes and associated application programming interfaces (APIs).

Ethereum blockchain explorers are used for tracing blocks and transactions using

queries. Some usage examples include finding all the information about a specific block/transaction, all transactions in a specific block, and what transactions were made to/from specific account-address. Some implementations of Ethereum blockchain explorers include: ERC20-Exporter [55] - a lightweight explorer that looks into all information on-the-fly from a back-end Ethereum node. It was developed with Node.js, Express.js and Parity. ERC20 [56] provides a common list of rules for Ethereum tokens to follow within the larger Ethereum ecosystem, allowing developers to accurately determine interaction between tokens. These rules include how the tokens are passed between addresses and how data within each token are accessed. ERC20-Exporter is used to explore the ERC20-based Ethereum tokens and supports Parity back-end node (the authors state it also supports the Geth client although this was not tested yet [55]). Initial data export for large tokens takes up to 30 minutes, as it tries to scrape the blocks' info like Ethereum Scraper [62] that exports the blockchain data by indicating start and end block number. EthExplorer [63] is a work in progress explorer developed with Node.js. Etherscan [23], ETCEXplorer [57], and Ethplorer [65] provide web-based UI and support mostly RESTful APIs, such as *getTopTokens* and *getTokenHistory*. They are implemented by calling basic methods on the Ethereum nodes and each implementation enforces its own limitation. For example, in Etherscan the API requests are limited to 5 requests/sec.

Etherchain Light [58], another lightweight blockchain explorer built with Node.js, Express.js and Parity, retrieves information on the fly from a back-end Parity node. It has extended the Ethereum Web3 API to provide some statistical measures such as transaction count and is still under development. Ethereum Explorer [59] is a decentralized client for Ethereum that interacts with the Ethereum blockchain via the Ethereum Web3 API, and provides users with basic interfaces to explore blocks. EtherQL [93] implements a query layer for Ethereum that supports some powerful APIs, e.g., range query and top-k queries and is backed by MongoDB as

the persistence layer to store blockchain data. vChain [138] proposes a solution to produce privacy preserving boolean query results in blockchain. The query result is paired with a cryptographic proof to guaranty its integrity. To support this verifiable query processing, vChain requires modification of the block structure to incorporate an authenticated data structure. To optimize query efficiency, inter-block and intra-block indexes are implemented.

Chapter 4

Generic blockchain-based provenance-aware system

In this chapter we discuss how our proposed approaches culminate to enable a generic blockchain-based provenance-aware system that can address blockchain traceability (regulatory) and security issues. To enable these capabilities, the system efficiently collects and manages comprehensive blockchain data provenance at the execution level, across time, and on-chain (as part of the consensus protocol). This enables efficient access to the data provenance by smart contracts, empowering them with richer online capabilities. Further, the system enables to efficiently query its offline data remotely, with privacy preservation support. Fig. 4.1 shows the aspects of the system that are addressed by our proposed approaches (in red). To help illustrate how the approaches complement each other to better address the cross-cutting concerns presented in the problem statement (section 1.1) we will reference Fig. 1.1 (Contributions and the cross-cutting concerns they address and supplement).

As discussed in the introduction, current blockchains innately trace data at the transaction level, e.g., from-address, to-address, Ether transferred. In the case that the to-address is a smart contract, the details of the inner contract flow operations are

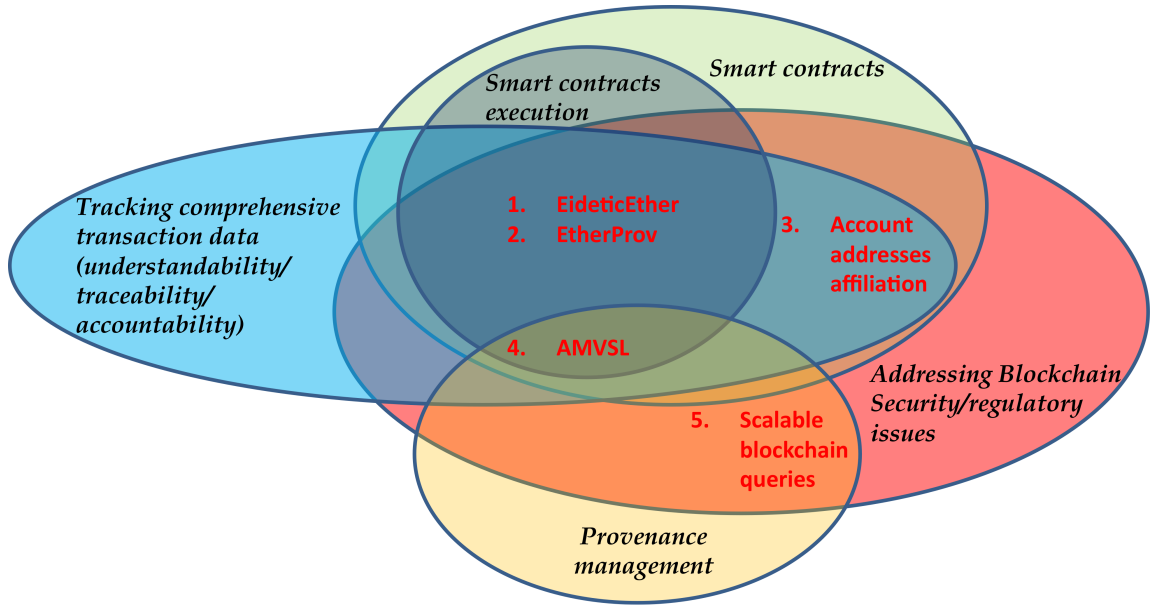


Figure 4.1: Mapping between the blockchain-based provenance-aware system aspects and our proposed approaches

not traced or stored by the blockchain, e.g., the contract execution flow, intermediate storage states' change, inner contract calls, or inner Ether transfers. In addition, when a transaction modifies the blockchain's state to a new state, the old state is not preserved. Enabling analysis, detection, and mitigation of security issues in Ethereum contracts, as well as richer traceability capabilities, requires the extraction of comprehensive blockchain provenance data comprising of execution flow and historic blockchain state, at each point in time. In the following sections we discuss how our proposed approaches enable to collect the above comprehensive provenance data, efficiently manage it, and use it to provide detailed tracing for regulatory purposes and enable rich security analysis, mitigation, and prevention of security issues.

4.1 Addressing security issues through provenance analysis in blockchains with smart contract support

In approach 1 (Chapter 5) we present the EideticEther framework that utilizes dynamic provenance to enable tracking historical contract execution flow that is composed of calling contracts' dependencies graph and their affected data provenance across time. We use this framework to explore how these capabilities can facilitate root-cause analysis and traceability. The type and granularity of the collected provenance dictates the extent of its derived applications capabilities. While EideticEther enables analyzing which contract call contributed to the change of specific storage data, it lacks a more detailed explanation on how this was achieved. Subsequently, its ability to facilitate efficient forensic analysis of suspected contract attacks and detect security issues is limited. By collecting more detailed provenance data, such as the contracts' execution trace, we can gain a more detailed understanding of the process, which can result in more capable and efficient root-cause analysis, traceability, as well as forensic analysis. Further, EideticEther's use of dynamic provenance limits its scope to executed paths. Since covering all possible paths is not usually feasible due to the path explosion problem, most approaches utilize static analysis [38, 66, 133] to abstract all possible execution paths without running the contract code, at the cost of over-approximation. Subsequently, to enable more efficient security analysis both static and dynamic analysis should be supported. Furthermore, when any security, defect, anomaly, or traceability issue is detected there should be a way to rectify it. This can prove challenging in blockchain due to the immutability of deployed contracts. To achieve these, in approach 2 (Chapter 6), we propose EtherProv, a comprehensive contract security analysis framework. EtherProv collects static provenance from contract Solidity source codes in the form of logic relations that include

their exact control flow graph (CFG). In addition, EtherProv collects dynamic provenance in the form of detailed and accurate CFG execution paths. Both provenance types comprise all interacting contracts using Solidity constructs. The dynamic CFG execution paths are collected using an efficient path profiling approach. EtherProv leverages the detailed static and dynamic provenance to efficiently detect known contracts' vulnerabilities, perform forensic/root-cause analysis of contracts, mitigate unaddressed security issues, and enable more detailed traceability capabilities, across all interacting contracts and across time.

Once a security issue is detected, and possibly mitigated, there may be a need to investigate other transactions that might have been affected by the attacker. However, in blockchain achieving this is challenging due to the pseudo-anonymity premise, which is essential to preserve the users' privacy on their publicly available transactional data. In approach 3 (Chapter 7), we propose to link account addresses that are used to deploy contracts written by the same developer. In our approach, we leverage stylometry techniques, widely used in the social science field for attribution of literary texts to their corresponding authors. The assumption underlying authorship attribution is the existence of a distinctive writing style, unique to an author and easily distinguishable from others. Drawing an analogy between literary text and contracts' source code, we explore the extent to which unique features of source code and bytecode of Ethereum contracts can represent the coding style of contract developers and be used to affiliate their deployers' addresses.

4.2 Efficient blockchain provenance management

In order to facilitate the analysis of security issues, defects, anomalies, and enable comprehensive traceability capabilities, additional provenance is required such as more complete execution flow of transactions and contracts, and their historical data,

across time. Current blockchain systems offer little to no support for efficiently managing and querying historical data. Archive nodes enable storing transaction execution traces and their related read and written storage values. However, these provenance data are not stored efficiently and do not provide for efficient querying. Further, the provenance is stored offline in a way that does not guarantee its authenticity or tamper-resistance. Ensuring authenticity and tamper-resistance is essential to provide rigid assurance of data when queried by external entities to the blockchain. In addition, since the provenance data are not synchronized by the blockchain nodes they cannot be consumed or produced by contracts online. Enabling contracts to access this provenance data can result in richer online processing capabilities and higher security assurance. Retaining additional historical provenance data in addition to existing data that are rapidly increasing in volume can result in high storage and computation costs. Since contracts are limited by gas, querying large amount of historical provenance data may not be feasible with current methods; therefore, its efficient management and querying is required. To address these challenges, in approach 4 (Chapter 8), we propose an on-chain authenticated and tamper-resistance index structure called Authenticated Multi-Version Skip List (AMVSL), designed to efficiently manage and support a rich set of query features over historical blockchain data. AMVSL is designed to operate alongside or replace current blockchain indexes and participate in the consensus protocol; therefore, it can be used to enable external entities to safely and efficiently query its data and enable contracts to query, as well as modify, its data. AMVSL enables retrieving authenticated data over a large range of keys and their current or historical values.

Regulatory bodies require frequent access to possible multiple blockchain nodes. However, since blockchain data are rapidly increasing and immutable, blockchain nodes consume a large amount of storage. The space requirement is magnified considerably in archive blockchain nodes that contain all historical data; therefore, it

is not feasible for the auditors to maintain their own nodes. This issue can be alleviated by enabling the auditors to remotely query existing nodes. However, since private blockchains may contain sensitive data, it is essential to preserve the privacy of the query results as they travel across company boundaries. Additionally, in order to ensure adherence to rigid regulatory assurance, the privacy of the auditors and their queries should also be preserved. As the queried data may require large amounts of data, efficient data retrieval is also required. In approach 5 (Chapter 9) we propose a system that enables multiple auditors to perform richer queries over remote blockchain data using big data solutions. The system additionally supports private information retrieval by utilizing cryptography techniques over semi-trusted servers to protect the auditors' identities, queries, and results. To handle the rapidly increasing blockchain data volume, the system employs a scalable distributed processing solution for big data. The proposed system is complimentary to our AMVSL approach. Indeed, combining these two approaches can allow auditors to remotely and efficiently perform richer queries over historical provenance data that are assured to be authenticated and tamper-resistant, while preserving the privacy of their identities, queries, and results.

Chapter 5

EideticEther: towards eidetic blockchain systems with enhanced provenance

With the wide adoption of contracts, their security emerged as a critical concern [48, 92]. As any software code, contracts are susceptible to security vulnerabilities and exploits, which are further fueled by the immaturity of the field. At the same time, companies are increasingly adopting blockchains with contract support to manage valuable assets, including crypto-currencies, securities, real estate and valuable tangible assets. Hence, failing to efficiently and promptly address issues relating to contract security, defects, anomalies, and traceability can result in hefty financial consequences. For instance, it has been reported [64] that a software bug involving the replacement of += operation with =+ resulted in the loss of assets worth \$800,000. In another incident [125] involved an attacker exploiting a defect in contract code to cause a \$80 million loss.

A blockchain system that can efficiently manage the provenance of both data and contract execution flow can be used to facilitate forensic analysis to help investigate

malicious activities, root-cause analysis to help investigate defects and anomalies, and support detailed traceability in the contracts. However, existing blockchain-based provenance solutions, such as the study by [118], do not track how the contract states evolved, for instance, which contract executions mutated these states; therefore, they are not suitable for the aforementioned requirements.

Devecsery et al. [50] discussed software systems that remember all operations, function calls, and states at any time and refer to them as *Eidetic software systems*. This historical information, or *provenance*, facilitates analysis of meta data, e.g., it enables queries that can answer questions about what states or calls affected other states or calls, and conversely what states or calls were affected by other states or calls. Inspired by this, we propose EideticEther, an *eidetic blockchain framework* that supports provenance for both data (blockchain states) and control flow (contract calls). EideticEther captures the provenance of executed contract calls flow (How-provenance), their parameters, and the relevant blockchain states before and after each contract call (Why-provenance). This provenance information can then be used to query the execution flow across time in different granularity levels, facilitate more complete understanding of the contracts' execution environment, and ultimately assist in better traceability, quality, and maintenance management.

5.1 Smart contract execution flow provenance

Existing approaches to capturing data provenance on blockchain [118] only gather information regarding the past history of blockchain states. While, this is quite useful, for the purposes of finding bugs in the contract code, or performing forensic analysis, more detailed information is necessary regarding how these states were changed through contracts execution.

Accurately tracking the information flow in the blockchain, which includes its states

along with the contract execution flow that operates on the data, can help identify the root cause of anomalies, debug defects in contract code, perform forensic analysis, and enable more detailed traceability for audit tracking purposes.

To this end, we present the EideticEther framework that enables to extract execution provenance across contract calls, their parameters, and their states before and after the calls. This is done by modifying the application programming interface of the software development kit that is used to write smart contracts to emit provenance information on run-time without affecting execution logic.

In the next sections we provide a motivating example relating to the retail sector and then describe our EideticEther framework. We conclude with an additional scenario that demonstrate our approach in the health insurance sector.

5.1.1 Motivating retail sector example

Alice, Bob, and Carol are friends and uPhone enthusiasts. Alice logs to an online retailer site to look for recent deals on the new phone model. She receives a list of suppliers that offer the phone alongside its price and quantity per supplier. Alice then picks the supplier that offers the best price and issues an order request. The sales department receives the order request, and issue an order request of their own to the suppliers department. The suppliers department issues a shipping request to the shipping department for the product from the specific supplier to Alice. The shipping department then issues a shipping identifier and returns it to the suppliers department that returns it to sales, that returns it to Alice. When the product is delivered, the shipping department sends the shipping confirmation to the suppliers, that send it to sales. Sales charge Alice using her recorded payment information and issue a payment confirmation for the suppliers, which issue a payment confirmation for shipping.

Alice then informs Bob about the supplier with the great deal she found. Bob logs

to the site, issues a similar order request, and calls to update Carol, who logs to the site but can not find the deal. He calls the customer service to inquire why.

The use case is composed of four scenarios:

1. Suppliers request - the user queries the sales department for a list of suppliers that sell a specific product. The sales department forwards the request to the suppliers departments and returns the retrieved list of the suppliers that have the product, its price and quantity.
2. Order request - The user selects a specific supplier and issues an order request for the product and its quantity from the selected supplier and sends it to the sales department. The sales department records the order in their blockchain state and forwards the order to the suppliers department. The suppliers department record the order from the sales department in their state and issues a shipping request to the shipping department. The shipping department returns a shipment identifier to track the order to the suppliers department, that forwards it to the sales department, which in turn returns the shipping identifier to the client. All departments update the shipment identifier in their corresponding state.
3. Shipping confirmation - upon receiving a shipment confirmation, the shipping department sends a shipment confirmation to the suppliers department, which sends it to the sales department. All departments update the shipment confirmation date in their corresponding state.
4. Process payment - only after the client received the product they issue a payment request to the sales department, which issues a payment request to the suppliers department, which in turn issues a payment request to the shipping department. All departments record the payment confirmation id and the payment date in their corresponding states.

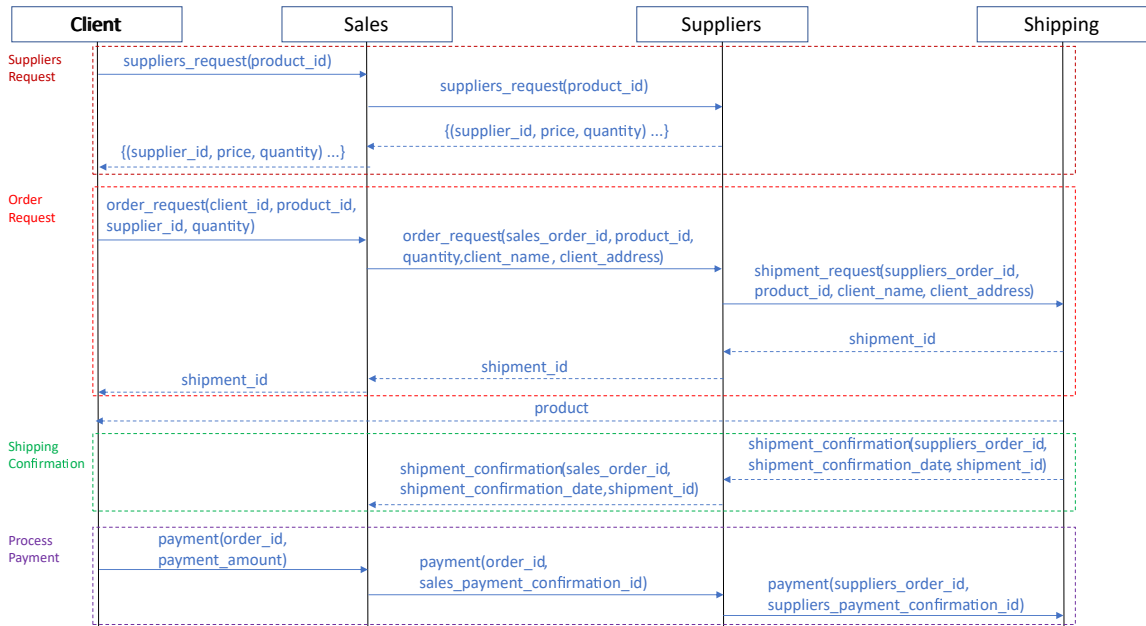


Figure 5.1: Execution flow of a full order request session

Fig. 5.1 provides the execution flow for all scenarios.

5.1.2 Implementation details

Each department's operational logic can be encapsulated by a contract, i.e., sales, suppliers, and shipping each has its own contract. Fig. 5.2 presents the scenario's implementation overview, which will be further discussed below.

Each department is represented by a contract that is used to issue and receive re-

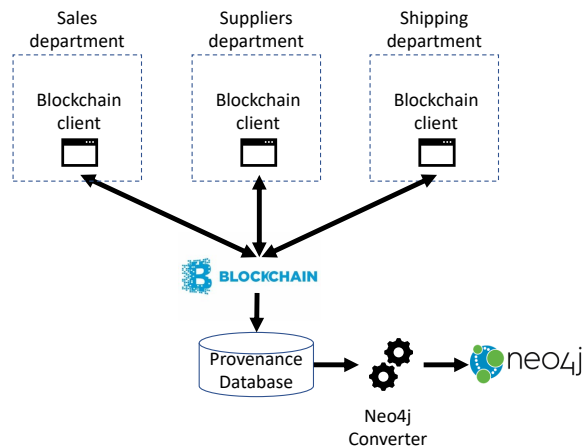


Figure 5.2: Scenario implementation overview

Contract	Function	Description
sales	order_request	Records the order details in the client_orders.DB and calls the order_request function at the suppliers' contract. The response should contain a shipment_id, which is updated in the order details.
suppliers	order_request	Records the order details in the sales_orders.DB and calls the shipping_request function at the shipping contract. The response should contain a shipment_id, which is updated in the order details and returned to the calling contract.
shipping	shipping_request	The client physical address details are updated, a shipment_id is generated and returned to the calling contract.

Table 5.1: Partial contracts description

client_id	client_name	client_address	payment_id
1	Alice	North st. 123	111
2	Bob	East st. 123	222
3	Carol	South st. 123	333

Table 5.2: Clients data

quests and maintain its state on the blockchain. Appendix A.1.1 contains the details of the model database schema for each department's contract.

Each department issues a contract call using its corresponding blockchain client, e.g., the sales department uses their own client to call the sales contract. The human client that issues the order request uses its own client.

A description of selected contract functions that participate in the order_request scenario can be seen in Table 5.1 (a more detailed description is available in Appendix A.1.1).

Transactions that are related to a specific scenario share a unique scenario_id in addition to a session_id that identifies the client's session, which is composed of different scenarios.

In our example, two user clients issue an order request: Alice with client_id 1 is the first to issue the order request for product_id 100 from supplier_id 2, which has only 2 units of the product. Bob with client_id 2 follows with an order request of the same product from the same supplier and orders an additional unit. Table 5.2 shows the

supplier_id	product_id	product_quantity	product_price
1	100	3	135
2	100	2	110

Table 5.3: Suppliers initial data

clients database and Table 5.3 shows the initial suppliers database.

5.2 The EideticEther framework

5.2.1 Collecting provenance data

EideticEther collects contract calls' execution flow and data provenance during runtime. This is achieved by modifying the contract execution infrastructure, namely the application programming interface of the software development kit that is used to write smart contracts, to include a non-intrusive provenance collection mechanism that collects the contract calls' execution information during run time and is composed of:

- Contract calls' graph.
- Execution parameters, such as the caller's (user or contract) name/address, the callee's (contract) name/address, and the API function and parameters to be invoked by the callee.
- Read/written states before and after the caller calls the callee.

When a caller calls the callee contract, only the states that are read or written by the caller and the callee are collected. This is done by modifying the blockchain context's get/set functions to record what addresses were read/written. The produced results provide a more concise view of the provenance and facilitates an easier analysis. The collected data are stored in a provenance database. A different process converts and exports the collected provenance data into the graph database such as neo4j [10] to enable visualizing and querying the provenance using the graph's query language.

EideticEther generates two types of provenance graphs: *contract calls' graph* - new nodes are generated and connected per contract call to reflect the provenance of the call graph. An example of this type of graph can be seen in Fig. 5.3; and *contracts'*



Figure 5.3: A session's contract calls' graph

parameters and state change graph - for each contract call its relevant parameters and the states before and after the call are captured and are connected to the contract's node. An example of this type of graph can be seen in Fig. 5.4.

5.2.2 Querying provenance data

The contract calls' graph provides information on what contracts were called during each session and scenario and by who (client/contract). Fig. 5.3 shows the contract calls' graph of Alice's session. From this we can see whether all contracts were called in the correct sequence. Each node contains an entity id and name, the session id, the scenario id, and the contract API function that was invoked on the callee contract. To get more detailed information on each entity call, further drill down is possible for each node by querying the contracts' parameters and state change graph, as we discuss shortly. Further, this graph can be used to answer queries regarding changes in different granularity levels. Following are query examples for different granularity levels.

Querying changes across multiple sessions Returning to the motivating example, Alice and Carol both ordered the new uPhone from supplier id 2. Carol could not find the same supplier and called the customer service to inquire about

Request datetime	Command	From client id	To	Supplier id
2020-01-10 2:02:34	order_request	1	sales	2
2020-01-10 3:16:15	order_request	2	sales	2

Table 5.4: Client order state changes across session - contract call parameters

Supplier id	Client id	Product id	Product quantity before request
2	1	100	2
2	2	100	1

Table 5.5: Client order state changes across session - API function parameters

it. A customer service representative queries the suppliers database and sees that the supplier with id 2 has no more products. To reveal the history that led to this current state, the representative can issue a query that shows what order requests, involving supplier with id 2, were issued and their details.

The query result was split into three tables (due to page space constraint): Table 5.4 - Table 5.6. Table 5.4 shows that customers with id 1 and 2 issued an order_request on the specific date and time for a product from supplier_id 2. The first line in Table 5.5 shows that before Alice and Bob issued the request the product's quantity was 2 and 1 respectively. Table 5.6 shows the supplier's remaining quantity, after the orders of Alice and Bob, was changed to 1 and 0 respectively.

Querying changes in a session across multiple scenarios Assume that Alice would like to know when her account was charged for the item. The query result for Alice's specific session in the sales department's client_orders_DB was split into three tables (due to page space constraint): Table 5.7 - Table 5.9. In Table 5.7 the first line shows that following the order_request the sales department created a client order row with id 111. In Table 5.8 the first row shows the parameters that are used

Supplier id	Product id	Product quantity after request
2	100	1
2	100	0

Table 5.6: Client order state changes across session - suppliers database row after contract call

Request datetime	Command	From	To	Sales order id
2020-01-10 2:02:34	order_request	sales	suppliers	111
2020-01-13 1:13:11	process_payment	sales	suppliers	111

Table 5.7: Client order state changes across session - contract call parameters

Sales order id	Supplier id
111	2
111	2

Table 5.8: Client order state changes across session - API function parameters

in this request. Since the shipping id is provided at the end of the order_request call it can be seen in the last column of Table 5.9. The second line shows that the payment_confirmation_id was updated at the end of the process_payment call.

Querying changes in a specific scenario and entity Fig. 5.4 shows a drill down query result for the suppliers node in the order_request scenario (marked by a red square as in Fig. 5.3), where the suppliers department receives the shipping confirmation for the product ordered by Alice from the shipping department. The middle node represents the suppliers department entity (i.e. contract). The attached green node represents the parameters used in the contract call such as the contract and the API function names, and the request datetime; the attached pink node contains the function specific parameters such as Alice’s name and address, and the suppliers’ order id; the blue node represents the relevant blockchain state before the shipping contract call; and the yellow node represents the relevant blockchain state after the call. As can be seen, the topmost nodes (marked by a black rectangle) are present only in the state after the call and represent the suppliers’ shipping order for Alice that was recorded through the shipping contract.

Sales order id	Payment confirmation id	Shipment id
111		555
111	123	555

Table 5.9: Client order state changes across session - client orders database row after contract call



Figure 5.4: Suppliers contract calls Shipping_request on Shipping contract

5.3 Health insurance sector example

In this section, we provide an additional scenario that illustrates how EideticEther can be used in the health insurance sector to provide root-cause analysis. In this scenario, patient Y has physiotherapy (PT) insurance, which was issued by an insurance company. When a PT clinic X sends an insurance claim to the insurance company for insurer Y, the insurance company declines the claim since the deductibles remaining for insurer Y's PT services are depleted. However, according to insurer Y's policy the deductibles should contain additional funds. Insurer Y calls the insurance company and requests an explanation.

5.3.1 Implementation details

As in the previous example, the participating service providers and the insurance company share a private blockchain. Each insurance service benefits in the insurer Y's policy is maintained in a separate contract. For example, the PT insurance

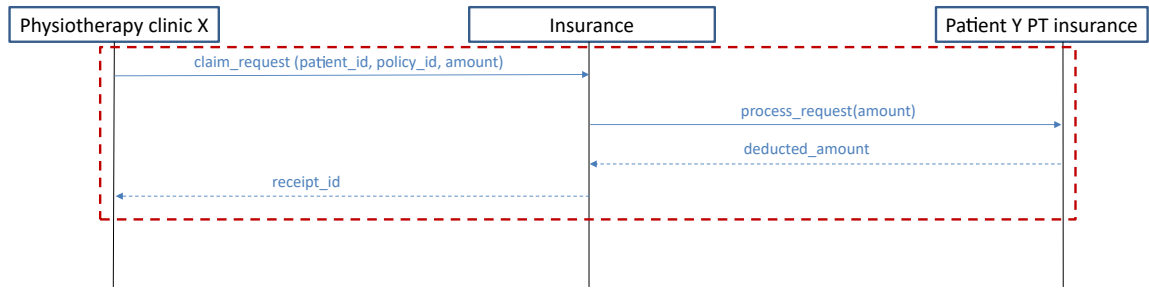


Figure 5.5: Insurance scenario execution flow

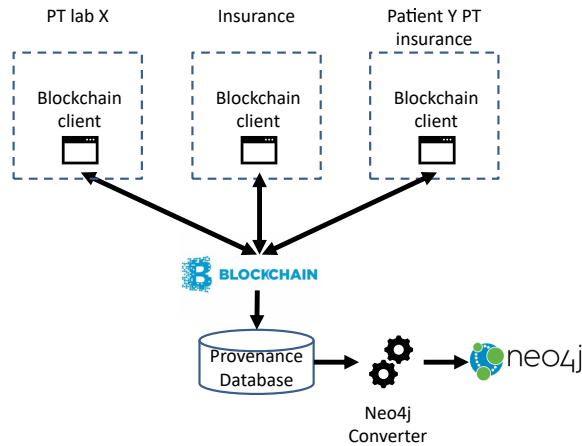


Figure 5.6: Scenario implementation overview

benefits that are associated with the insurer Y’s policy are maintained in a contract called “Patient Y PT insurance”. This contract contains the account membership type (*account_type*), which can be either “silver” or “gold”, and a deductibles field containing the amount that is left to claim by insurer Y according to their policy. The insurance company employs a single “insurance” contract to manage all insurers and their service benefits’ contracts. Each service provider is represented by a single contract, e.g., PT clinic X is represented by the “PT clinic X” contract. Appendix A.1.2 contains the details on the model database schema for all contracts. When a service provider, e.g., PT clinic X, sends a claim request through the “PT clinic X” contract, with the provided claim details, to the “insurance” contract for a specific insurer’s policy, the “insurance” contract calls the insurer’s service benefits contract with the deductible amount. Fig. 5.5 provides the execution flow overview for the insurance scenario. Fig. 5.6 presents the scenario’s implementation overview.

Contract	Function	Description
PT clinic X	claim_request	Receives the patient details and forwards it to the insurance contract using the claim_request command. Returns the receipt including the deducted amount.
Insurance	claim_request	Receives the patient details and the provider's contract address (from the contract call). Locates the provider details to ensure the provider is registered in the providers_DB. Opens a request and stores it in the requests_DB. Using the received patient_id, policy_id, and field_id it extracts the contract_address from the patient_contracts_DB. The contract_address is used to call the patient_y_pt_insurance contract with the amount to deduct. Returns the receipt including the deducted amount.
patient_y_pt_insurance	process_request	Receives the amount to deduct, deducts the maximum available amount, and returns the actual deducted amount.

Table 5.10: Health insurance - contracts' API description

Appendix A.1.2 contains the details on the model database schema for all contracts. A description of selected contract functions that participate in the claim_request scenario can be seen in Table 5.10 (a more detailed description is available in Appendix A.1.2).

5.3.2 Querying provenance data

The insurance company's customer representative issues a query that retrieves all claim requests made to the "Patient Y PT insurance" contract. The query results are provided in an ascending order. Due to space constraints, we present the results in two separate tables: Table 5.11 contains the contract call parameters that were used by the providers' contracts when calling the "insurance" contract. The "From" field contains the details of the provider and the "To" field contains the insurer Y's contract that was used in processing the claim; Table 5.12 contains the corresponding details that were used in the claim requests submitted by the provider. The "Amount" field contains the claim amount and the "Patient Y PT insurance deductibles after contract call" field contains the remaining deductibles amount after the corresponding contract call. The customer representative sees that (1) the current contract's deductibles are indeed depleted, and (2) in the first request, the initial deductibles amount can be computed as $80 + 320 = 400$, which is the allo-

Request datetime	Command	From	To
2020-01-10 2:02:34	claim_request	PT clinic A	Patient Y PT insurance
2020-01-17 2:01:10	claim_request	PT clinic A	Patient Y PT insurance
2020-01-24 4:00:12	claim_request	PT clinic X	Patient Y PT insurance
2020-01-31 4:06:33	claim_request	PT clinic X	Patient Y PT insurance
2020-02-07 4:01:22	claim_request	PT clinic X	Patient Y PT insurance

Table 5.11: Insurance scenario - contract call parameters

API function parameters				Patient Y PT insurance deductibles after contract call
Patient id	Policy id	Provider id	Amount	
112334	2	200	80	320
112334	2	200	80	240
112334	2	100	80	160
112334	2	100	80	80
112334	2	100	80	0

Table 5.12: Insurance scenario - API function parameters and deductibles after contract call

cated amount for the “silver” account_type. However, the customer had purchased the “gold” insurance policy, which entitles them to a 800 deductibles. Hence, the issue has been determined to be the result of human error.

5.4 Comparison to related work

Ruan et al. [118] provide an approach that enables efficient access of historical data from within a contract, however it does not track what initiated the changes and how. EVM* (Section 3.2.2) is the system that is most related to EideticEther. EVM* requires the modification of a full node and enables extracting dynamic provenance in the bytecode level on run-time. However, as it provides online attack detection, it enables extracting only data provenance that is consumed online, which is not further stored. In contrast, EideticEther is designed to extract and store data provenance as well as function call dependencies provenance, using the high level constructs that are used in the contracts’ programming language, which enable more efficient and detailed traceability and root-cause analysis.

Chapter 6

EtherProv: provenance-aware detection, analysis, and mitigation of Ethereum smart contract security issues

In chapter 5 we discussed how execution flow provenance can enable traceability and root-cause analysis capabilities. In this chapter we propose a framework that captures a more comprehensive and detailed level of provenance, which enables, through finer grained traceability, to detect contracts' known security issues, provide efficient and timely forensic and root-cause analysis of unaddressed security issues, and provide mitigation of compromised transactions. The majority of the current security verification approaches that address *known* security issues leverage static analysis of source code [66, 78, 83, 130] or bytecode [38, 133].

A contract's Solidity source code contains important information that is lost in compilation to EVM bytecode. This information enables security experts, auditors, and developers to detect and analyze vulnerabilities and attacks, which would be

challenging or impossible to analyze or detect with the compiled bytecode alone. For example, the information of values' types is required to detect the integer underflow/overflow vulnerability; in addition, the logical data structures used in the Solidity source code, e.g., composite structures, arrays, and mappings help to logically group detached storage addresses to more coherent structures that help the rationalization of the program's overall logic. As, these are lost in the compilation process, the program analysis becomes considerably more cumbersome. Providing this information can help provide for more accurate, efficient, and timely analysis. In addition, bytecode analysis incurs imprecise resolution of memory and storage offsets when identifying variables [133], and jump targets when identifying stack locations statically [38], which may lead to higher false positive/negative rates. Hence, a control flow graph (CFG) that is extracted from the Solidity source code and preserves its constructs enables more capable and efficient analysis compared to a CFG that is extracted from the bytecode.

Static analysis operates on contracts' source code before they are run, and primarily aims to discover the presence of potential code security vulnerabilities. Understanding whether these vulnerabilities are exploitable requires an analysis of the contracts' execution flow, and hence, the execution of the contracts' bytecode. The large-scale analysis of contracts conducted by Perez and Livshits [109] found that only 2% of vulnerable contracts are exploitable.

Contracts can be called by external accounts, as well as by other contracts; therefore, dynamic parameters (e.g., function call parameters and current storage states), externally called contracts and libraries, and transaction data, should all be taken into account while evaluating the vulnerability and exploitability of the contracts.

Current approaches for addressing *new* security issues use dynamic analysis on transaction execution. While these approaches enable a fine grained data collection and analysis at the function/instruction level, they require to collect huge amounts of

execution traces in order to achieve large coverage, which requires considerable computation and storage resources that can limit its applicability. Since in the dynamic analysis approach only executed paths can be analyzed, vulnerabilities that are contained in unexecuted paths are not possible to detect [143]. Additionally, current dynamic analysis approaches require the modification of a full node with considerable instrumentation [45, 52, 77, 98, 115, 131, 143].

Since on the one hand, static analysis can achieve complete path coverage at the cost of higher false positives, and on the other hand, dynamic analysis can provide higher precision at the cost of fewer covered paths, a comprehensive solution that enables both static and dynamic analysis can benefit from both worlds and help to analyze and detected more vulnerabilities and attacks.

Due to the immutability of deployed contracts, handling unaddressed security flaws or malicious behavior is challenging. For example, in 2016 an exception handling bug, which was discovered in the popular contract game “King of the Ether Throne” resulted in mishandled exceptions. To resolve the issue, the developers were forced to manually reimburse some players and publicly request players to not use the contract [8] in order to prevent further monetary loss. This mitigation approach, where a vulnerability is discovered after a contract’s deployment, is often the only feasible solution due to the immutability of the deployed contract. Once a contract is deployed, its code cannot be changed to patch unhandled security flaws or remove malicious behavior. Ma et al. [98] proposed an online reinforcement of compromised contracts through online bytecode dynamic analysis. To achieve this, the authors modified a full node’s EVM execution code. Although, this approach has several drawbacks. First, in blockchain, transaction executions are immutable; hence it is crucial to enable precise analysis that results in an accurate mitigation strategy. Any discrepancy may result in ever higher monetary losses. As discussed above, the accuracy of a CFG that is extracted from the bytecode is lacking compared to a

CFG that is extracted from the Solidity source code. Second, to be able to use this approach all full nodes are required to be modified with the enabling mechanism in order to participate in the consensus protocol.

As the rate of new security attacks increases, there is a need for an environment that can facilitate efficient detection and analysis and accurate and reliable mitigation of deployed contracts' security issues to reduce damage and monetary loss in a timely manner. To achieve this, it is necessary to collect fine-grained contract execution trace, which we call contract execution flow provenance or execution flow provenance for short. To this end, we propose **EtherProv**, a contract execution provenance tracking framework that leverages static and dynamic analysis synergy *to efficiently detect known contracts' vulnerabilities, analyze new security vulnerabilities and attacks, and mitigate unaddressed security issues across interacting Ethereum contracts, and across transaction history, in an accurate and reliable way.* To achieve this, EtherProv extracts static provenance from the contracts' Solidity source code that contains pre-compiled detailed information. The collected static provenance is used to provide an efficient path profiling inspired by Ball et al. [34] that accurately captures the execution flow path across multiple deployed contracts during contract execution. EtherProv makes use of contract bytecode instrumentation, which as opposed to the existing approaches, does not require full node modification and is transparent to the consensus protocol. When a deployed contract is executed, for each execution trace, EtherProv extracts a compressed encoding of the CFG's exact execution path that consists of no more than a few integers. This can alleviate the storage costs incurred when collecting a huge amount of traces. The compressed execution flow encoding is emitted to the blockchain, which can then be retrieved and decoded to the full executed path. The full executed path contains the exact executed CFG path comprising of detailed Solidity constructs. To enable a more thorough analysis, EtherProv collects online dynamic data, e.g.,

block/transaction ids, callers' addresses, function call parameters, and storage state values. The collected dynamic data and encoded paths are then transformed to a unified representation and are stored in a provenance database as Datalog facts.

Security analysis in EtherProv is based on the fact that while security concerns in contracts may be complex, once these issues are discovered and defined, they can be detected by checking a contract's compliance with the known security properties. To detect known security issues within a single contract and across multiple contracts, the provenance database is queried using Datalog *compliance queries* that are composed of high-level Solidity constructs.

Besides identifying known security issues and analyzing new security threats, EtherProv is capable of mitigating unaddressed security threats that are detected within a deployed contract. Since the exact execution paths and their static analysis data are known before the contract terminates, EtherProv enables a reliable and efficient dynamic modification of the control flow execution for specified execution paths during contract execution, e.g., reverting the contract call if a path with specific (e.g., undesirable) properties is encountered. To the best of our knowledge, none of the existing approaches support such mitigation with deployed contracts.

To evaluate EtherProv's capabilities, we show an analysis of three well known security vulnerabilities, i.e., Liquid Ether, Re-Entrancy, and Restricted Writes. In comparison with a well-known contract analyzer, Slither [66], EtherProv is capable of detecting all issues *across* contracts. We present a scenario to illustrate our approach's forensic analysis of unaddressed security concerns within already deployed contracts. We further evaluate EtherProv's mitigation capability on the original "King of the Ether Throne" contract. As our evaluation shows, EtherProv can efficiently identify vulnerable contracts with an average contract instrumentation gas overhead of 18.9%. To summarize, our main contributions are as follows:

- We propose EtherProv, the first comprehensive contract security analysis frame-

work that supports static and dynamic analysis across contracts and transaction history. The tracked provenance is composed of an accurate CFG containing Solidity constructs, which are easier to reason about and maintain compared to bytecode constructs. This is achieved using bytecode instrumentation that does not involve the modification of full nodes and hence is transparent to the consensus protocol. The code is publicly available¹.

- EtherProv provides an efficient path profiling approach inspired by Ball et al. [34] that accurately captures the CFG execution flow path across multiple deployed contracts during contract execution and with low gas consumption.
- EtherProv uses a unified data schema over Datalog logic relations composed of static and dynamic data. Logic-driven queries enable efficient detection of contracts' known security issues, efficient and timely forensic analysis of unaddressed security issues in executed transactions, and the extraction and incorporation of their pattern to be used in detecting future similar issues.
- EtherProv enables an efficient and reliable mitigation of compromised deployed contracts.

6.1 The EtherProv framework

EtherProv is a blockchain and contract provenance tracking framework that enables detection, analysis, and mitigation of security issues across interacting Ethereum contracts, and across transaction history.

The framework is designed (1) to provide tracking of transactions including detailed contracts' execution flow provenance through static and dynamic analysis and extend it to encompass an entire program's control flow spanning multiple functions'

¹<https://github.com/shomzy/EtherProv>

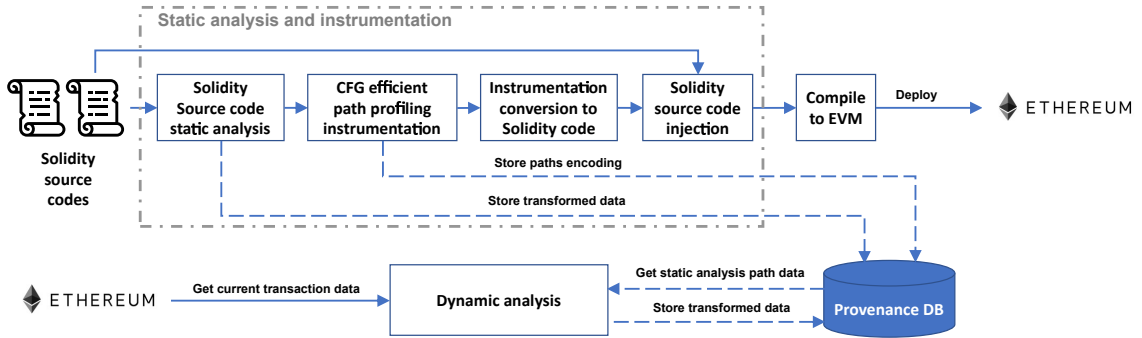


Figure 6.1: EtherProv Framework overview

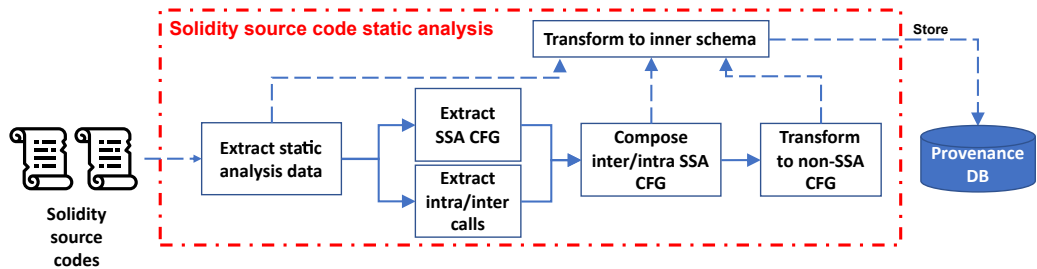
interaction across contracts; (2) to detect known security vulnerabilities, in a single contract and across interacting contracts, by checking compliance with known security properties; (3) to analyze unaddressed security vulnerabilities, in a single contract and across interacting contracts; and (4) to provide reliable mitigation functionality for transactions vulnerabilities in already deployed contracts. The overview of the framework is presented in Fig. 6.1.

Provenance collection Given a contract’s source code, EtherProv extracts information on control structures, storage manipulations, and function calls including their corresponding parameters from each function’s CFG. The individual function CFGs are then combined to construct an *extended CFG* encompassing the entire contract’s control flow spanning multiple functions’ interactions across contracts. The extended CFG is then used to encode individual paths and calculate an efficient paths profiling. The encoded paths, in addition to the CFG edges that comprise each encoded path, are stored in the provenance database. To facilitate tracking capabilities and collection of provenance data during contract’s execution, the contract’s source code is instrumented along the extended CFG’s paths. The instrumented contracts are then compiled into EVM bytecodes and deployed to the blockchain. Upon the contracts’ execution, the executed path’s encoding is emitted to the blockchain, collected off-chain along with additional dynamic data, and stored in the provenance

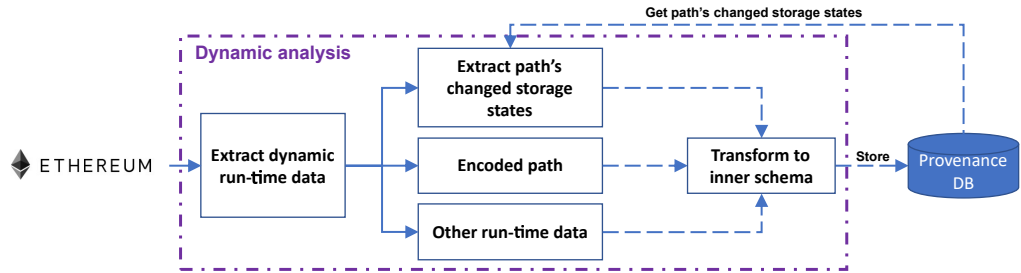
database.

Security analysis The information collected both statically and dynamically enables a more capable contracts' security analysis. More specifically, we check a contract's compliance with the security properties of known vulnerabilities and enable efficient forensic analysis of the exploitability of this vulnerability along the executed contract flow; across different execution flows of the same contract; and across different execution flows of different contracts across transactions. In this analysis, we leverage the Datalog language syntax.

Tracking & Mitigation The mitigation component is designed to accurately and reliably analyze a transaction execution in real time based on the collected provenance information. If a transaction execution violates security properties, EtherProv is capable of dynamically reverting control flow execution for specified execution paths in real time, effectively interrupting transaction execution.



(a) EtherProv detailed static analysis data extraction flow



(b) EtherProv detailed dynamic analysis data extraction flow

Figure 6.2: EtherProv detailed static and dynamic analysis data extraction flow

6.1.1 Smart contract execution provenance collection

EtherProv collects fine-grained contract execution provenance through static and dynamic analysis of the Ethereum contracts (Fig. 6.2).

6.1.1.1 Generating extended CFG

Given the contracts' Solidity code, the source code static analyzer extracts static analysis data (e.g., control structures, storage manipulations, function calls, etc.) from contracts, derived contracts, contract's functions, variables, interfaces, and libraries. Next, each contract's function CFG is extracted. The CFG is represented as a graph of nodes. Each node is provided in two forms of granularity: Static Single Assignment (SSA) form and non-SSA form. Each Solidity source code statement may be represented by a single non-SSA node. In contrast, the SSA form for the same statement may contain multiple SSA nodes. For example, for the single-line Solidity statement `uint variable = array[index];`, the non-SSA form representing the statement consists of a single non-SSA node. The SSA form is composed of at least two SSA nodes: one SSA node for storing the de-referenced array cell in the provided index in a temporary variable and a second SSA node for storing the temporary variable in the lvalue variable. EtherProv stores, for each contract, its Solidity CFG in both SSA and non-SSA forms. The finer granularity of the SSA form is used to analyze all commands constituting a statement, i.e., finding which variables are written-to/read-from in statement; identifying a function call, its destination address, and parameters; identifying statement type, e.g., IF, Expression, and Loop. The granularity of the non-SSA form includes the Solidity statements' code and location in the source code, which is used to instrument the Solidity source code. The mapping between SSA and non-SSA forms are also stored and used to provide static/dynamic analysis capabilities in either granularity. Individual function CFGs include information on calls to functions inside the same contract or to

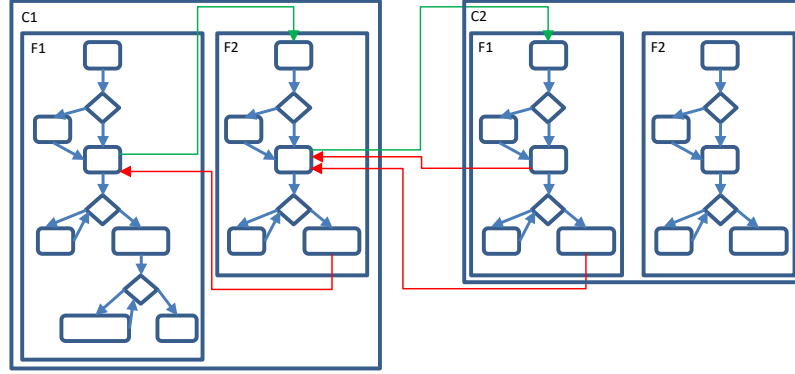


Figure 6.3: Extended CFG

functions in external contracts, when applicable, including constructor calls when inner contract instantiation occurs. In order to enable static analysis of an entire program’s CFG spanning multiple functions’ interaction across contracts, EtherProv extends each contract function’s CFG to encompass intra-/inter-contract calls. This is achieved by adding new edges for each SSA node if it is a call site. One edge is added from the call site SSA node to the first SSA node of the called function. Additional edges are added from each terminating SSA node, e.g., a return statement, in the called function back to the calling call site SSA node. Similarly, the non-SSA CFG is extended with the intra-/inter-contract function call edges. The resulting CFG, covering all functions of all contracts, is referred to as an *extended* CFG. Fig. 6.3 shows an example of an extended CFG of two contracts: C1 and C2, each containing two function CFGs. The green arrows represent calls from non-SSA call sites and red arrows represent flow returns from terminating non-SSA nodes. In the following we refer to an *extended* CFG as a CFG.

6.1.1.2 CFG efficient path profiling

In order to enable efficient tracing of contracts’ execution paths at run-time, EtherProv instruments the (extended) CFG using a path profiling approach inspired by Ball et al. [34]. The need for efficient path profiling stems from the significant overhead typically incurred by instrumentation, which can be unacceptable in the

Ethereum environment where contract execution consumes gas of a limited quantity. Ball’s algorithm gives an efficient approach to accurately determine the frequency of a control flow path. As part of the algorithm, the CFG is transformed to a directed acyclic graph (DAG) by rearranging the loops’ edges. A single “entry” node is added to point to all the possible “entry” nodes of the DAG and a single “exit” node is added to which all terminating nodes of the DAG are pointing. Further, an edge is added from the “exit” node to the “entry” node, effectively creating a single cycle in the “DAG”. The algorithm efficiently enumerates and uniquely encodes each possible path by assigning a single integer to each edge. When a path is traversed from the “entry” to the “exit” nodes, the values of the edges are summed to produce a unique path number. Following, A spanning tree is constructed from the “DAG”, where the edges, which are not part of the spanning tree are chosen to contain instrumentation to accumulate the traversed path’s encoding. Each instrumentation adds a single integer when traversed. For each instrumented edge, which is also defined by its “from” and “to” nodes, the instrumentation value is calculate by summing all edges values that are in the paths from the instrumented edge’s “to” node to its “from” node in the spanning tree, which results in a positive integer. When a path in the directed spanning tree is traversed in reverse the resulting instrumentation value is negative. The resulting instrumented edges, which are not part of the spanning tree, accumulate an equivalent path encoding result at the end of a path. To apply this algorithm in the context of contracts, several deficiencies need to be addressed.

First, a path in our context may span multiple contract interactions. To address this, EtherProv uses a `PathAccumulator` contract containing a single `int256` storage variable, which serves as the “global” *path accumulator* register (*accumulator* for short) and functions to manipulate it.

Second, each contract instruction consumes some amount of *gas*, a measurement unit used to calculate a transaction’s execution cost. Ball’s algorithm requires to produce

a path encoding per loop iteration, which is expensive in Ethereum context. In order to reduce gas consumption, EtherProv emits the path using the low gas consuming `emit` instruction, which writes the accumulated value to a transaction’s event. When the entire path is emitted, the accumulator is reset to 0, which refunds gas. Further, EtherProv reduces multiple log writes in a loop by compressing repetitive path encodings.

Third, a Solidity contract may have multiple entry point functions. These include *external* functions, which can only be called from outside the contract, or uncalled *public* functions. A contract may also have multiple exit points in the form of statements that terminate an entry function (e.g., return statements, loop breaks, uncaught throw statements). Since Ball’s algorithm is designed to work on a DAG, we convert the CFG to a DAG by connecting the single DAG’s “entry” node to the CFG’s nodes that represent the first statement of each entry function. The DAG’s “exit” node is connected to each CFG node that represents an entry function’s terminating statement. Further, each back-edge (e.g., resulting from while loops, recursive function calls) is replaced by two additional edges: one edge pointing from the DAG’s “entry” node to the back-edge’s destination node and one edge pointing from the back-edge’s source to the DAG’s “exit” node. Fig. 6.4(a) shows an example CFG and its corresponding DAG in Fig. 6.4(b). The corresponding unique encoding of each DAG path is shown in Fig. 6.4(c). Each path, which may be comprised of multiple edges, is encoded as a single value, resulting in a path compression. In order to calculate a path encoding during run-time, the generated DAG is instrumented with an accumulator. Each instrumented edge adds a positive or negative value along the taken execution path, as explained above. At the end of each path execution, the accumulated path’s encoding is emitted (logged) in the order it was encountered, which correlates to the order of the contract’s execution flow. The instrumented DAG edges are then copied to the corresponding CFG edges to form an *instrumented CFG*

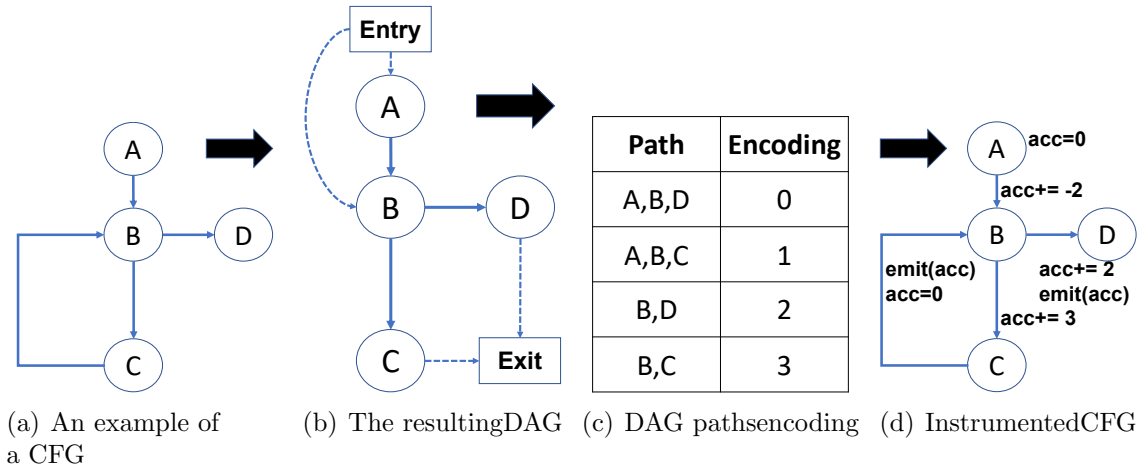


Figure 6.4: Efficient path profiling - CFG instrumentation and paths extraction

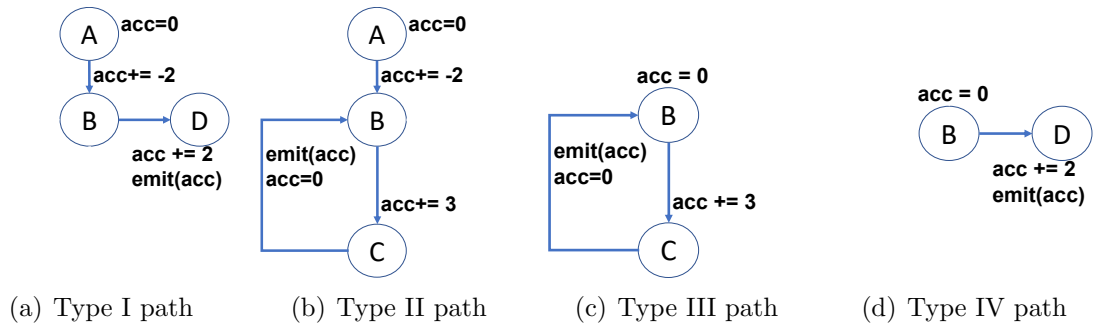


Figure 6.5: Efficient path profiling - illustration of four path types

(see Fig. 6.4(d)).

Fourth, an instrumented CFG may contain loops. Paths without loops are encoded as a single value. Paths containing at least one loop are encoded using multiple values. A back-edge in the instrumented CFG causes the current accumulated path encoding to be emitted, and the accumulator to be reset, in order to accumulate a new encoding of the paths that succeed the loop from that node. There are 4 path types: **I** - a path not containing a loop, **II** - a path starting at the beginning of the “entry” node and leading to the first loop, **III** - a path after a loop leading to another loop, and **IV** - a path after the last loop leading to the “exit” node.

Fig. 6.5 shows the 4 path types, which are extracted from the instrumented CFG example shown in Fig. 6.4(d). Fig. 6.5(a) shows a type I path not containing a loop.

The accumulator is incremented from 0 to -2 and back to 0 at the path's end, where it is emitted before node D, as the path encoding 0. Fig. 6.5(b) shows a type II path entering a loop for the first time. The accumulator is incremented from 0 to -2 to 1 at node C ($-2 + 3$), where it is emitted before node B as path encoding 1. The accumulator is then re-initialized to 0 for the accumulation of the next path's part, which can be another loop iteration, leading to a different loop, or leading to the "exit" node. Fig. 6.5(c) shows a type III path, which iterates through a loop. The accumulator is incremented from 0 (after it was re-initialized in the type II path) to 3 at node C ($0 + 3$), and is emitted before node B, as path encoding 3. The accumulator is then re-initialized to 0 for the accumulation of the next path. Fig. 6.5(d) shows a type IV path, which leads from the last loop iteration to the "exit" node. The accumulator is incremented from 0 to 2 at node D ($0 + 2$), and is emitted as path encoding 2.

When no loop is executed, a single path encoding is emitted. When a loop is executed, a path encoding is emitted for each iteration. To efficiently emit a loop iteration encoding (type III path) at run-time, EtherProv counts the number of loop iterations that were encountered and emits a single type III path encoding along with its count. For example, for a path not containing loops such as A-B-D, the emitted encoding is 0. For a path containing loops such as A-B-C-B-C-B-C-B-D, the encoding is 1 (for A-B-C), 3 (for B-C), 3 (for B-C), 2 (for B-D). The 2 loop iterations (B-C-B-C) are accumulated and emitted as the encoding for B-C along its count. The final compressed path is emitted as (1), (3:2), (2). The mapping of each encoded path to its instrumented CFG's related edges is stored in the provenance database as a Datalog fact.

```

constructor(uint256 p) public {
  wizardAddress =msg.sender;
  currentClaimPrice =startingClaimPrice ;
  currentMonarch = Monarch(wizardAddress, "[Vacant]", 0,
  block.timestamp);
}

```



```

constructor(address pathAccAddress, uint256 p) public {
  _path_acc = PathAcc(pathAccAddress);
  _path_acc.init_all_transaction_data(1030);
  wizardAddress =msg.sender;
  currentClaimPrice =startingClaimPrice ;
  currentMonarch = Monarch(wizardAddress, "[Vacant]", 0,
  block.timestamp);
  _path_acc.inc_transaction_path_acc(0);
  _path_acc.flush_path_data();
}

```

Figure 6.6: Solidity source code CFG instrumentation

6.1.1.3 Instrumenting Solidity source code

The instrumented CFG provides efficient and accurate profiling of CFG paths. To facilitate provenance collection during the contract’s execution, we translate each of the CFG’s edge instrumentations to their corresponding Solidity statements. For example, the accumulator initialization to 0 is converted to the Solidity statement `_path_acc.init_all_transaction_data(0);`.

Since each CFG instrumented node connects between two node, where each node represents a Solidity statement with a known location, EtherProv heuristically determines whether instrumentation should be injected after the source statement or before the destination statement. When mandated by the instrumentation, additional code is injected. For example, an *if* statement without an *else* clause, may have an instrumented edge corresponding to the *false* branch, in this case an *else* clause is injected with the instrumented code.

Additional dynamic execution data are extracted with further Solidity code instrumentation, which includes a getter function for each of the contract’s public/private storage state variables to enable to query their values. The modified Solidity source code is then compiled to EVM bytecode and deployed to the Ethereum blockchain. The accumulator that holds path encoding is instrumented through the use of the `PathAccumulator` contract, that is created beforehand. Its address is injected in each contract’s constructor in the original Solidity source code, in addition to code that saves the `PathAccumulator` contract in an internal contract storage state.

Fig. 6.6 shows an example of Solidity source code CFG instrumentation, where the

instrumentation statements are depicted in red.

6.1.1.4 Dynamic data extraction

EtherProv enables the extraction of an execution flow, consisting of high-level detailed Solidity code statements by instrumenting only the contract’s bytecode. The contract’s Solidity code is instrumented to emit an encoding of an exact execution flow path of a contract in real time to the transaction log. The encoded path can then be extracted from the PathAccumulator contract’s events in any historical transaction containing an instrumented contract. The contract’s exact execution flow path is decoded from the encoded path by querying the provenance database for the corresponding full path.

To enable more advanced analysis capabilities, EtherProv additionally collects different types of dynamic provenance data, i.e., the called function, its parameter names and types, and the contract’s name are extracted from the contract’s Application Binary Interface (ABI); block and transaction numbers are extracted from the mined block; from address, to address, called function name and parameter values are extracted from the mined block’s transaction.

To enable extracting each contract’s current storage state values and current Ether balance, EtherProv is run after each transaction to collect the current values, which may change in the subsequent transactions. The current contract storage variable values are extracted from the contract’s instrumented getters. To save computation and space, EtherProv queries the static analysis data from the data provenance database to: (1) decode the fully executed path from the extracted encoding, and (2) to extrapolate only the storage states that were read from or written into, according to the decoded executed path.

Appendix B.1 provides more detailed information on the EtherProv model and the static and dynamic collected provenance.

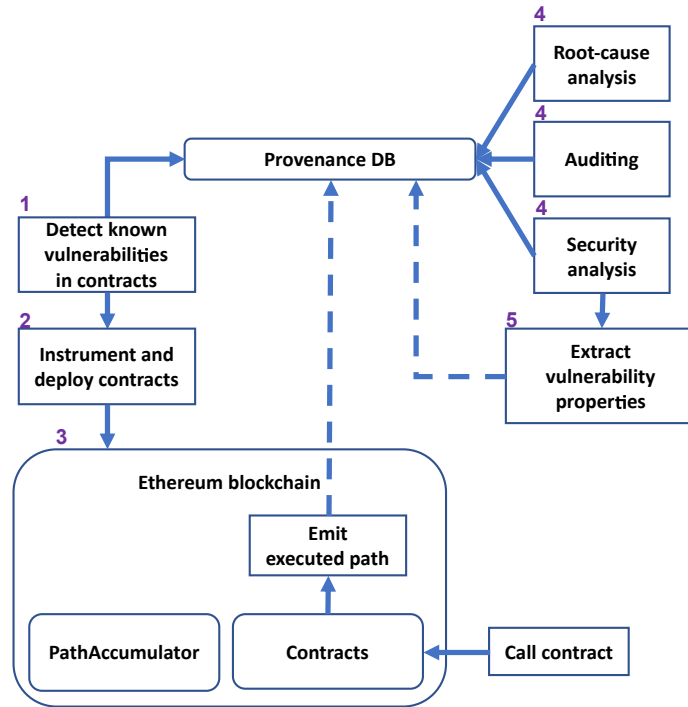


Figure 6.7: EtherProv security analysis flow overview

6.1.2 Security analysis

To enable security analysis and debugging, Datalog rules are issued against EtherProv’s provenance database. The provenance database is itself composed of an extensional database (fact tables) and intensional database (derived by rules). The fact tables are composed of static and dynamic analysis tables that are populated in the corresponding phases. Static analysis tables include information on contracts, their functions and parameter values, CFG nodes in SSA/non-SSA form, variables throughout contracts and their functions, paths and their related state access details, etc. Dynamic analysis related tables include mapping information between the called contract address to its static contract data, parameters used in the contract call, and the extracted path and its related read/written state parameter values. The intensional database comprises the security and debugging analysis rules, which use the extensional database (fact tables).

Fig. 6.7 shows the security analysis flow supported by EtherProv. We next explain

the flow in detail. (1) Before deploying the contracts, their static provenance data are extracted, stored in the provenance database, and analyzed for known vulnerabilities using the security analysis rules. If any vulnerability was detected, EtherProv generates a notification and stops further operation. Otherwise, (2) the contracts' bytecode is instrumented and deployed to the blockchain. (3) When a transaction that contains the instrumented contracts is executed, the dynamic provenance is collected using the instrumentation, emitted at the end of the transaction, and stored in the provenance database. (4) The stored provenance is used to perform root-cause analysis, debugging, traceability analysis, or security analysis. (5) If an attack was detected, a Datalog rules query that can detect the attack pattern is created and stored in the provenance database. When new contracts go through the EtherProv framework they are first checked against all existing vulnerabilities in the provenance database (step 1), which include the newly added detection patterns. In Section 6.5 we provide examples of EtherProv queries enabling root-cause, forensic, and security analysis.

6.1.3 Tracking & Mitigation

Since a deployed Ethereum contract is immutable, addressing undesired contracts' execution is challenging. EtherProv tracks the executed contract's path during contract execution and hence, knows its outcome before the transaction commits. The collected provenance data, e.g., current executed path's input and output storage state parameters, local and global variables, and function parameters, are used to identify and collect auxiliary data to help inform a mitigation action in real time. Such mitigation can modify the current execution flow, e.g., revert the current transaction or call additional internal/external functions, if the transaction's path is determined to be suspicious or malicious.

The paths corresponding to the known properties of undesired issues can be re-

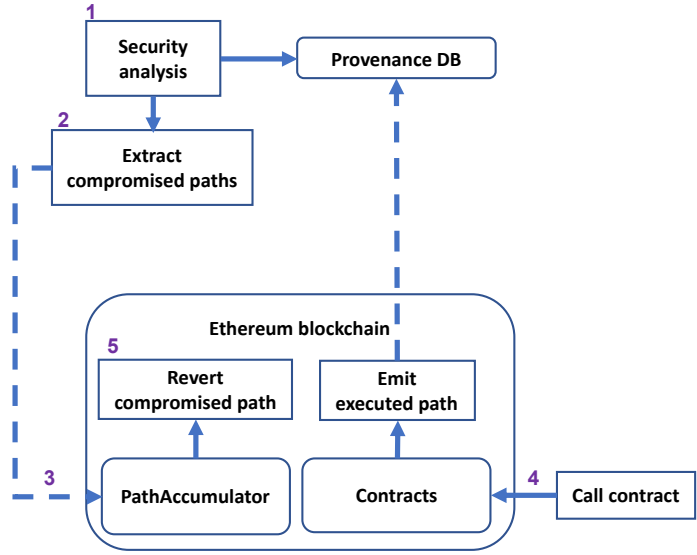


Figure 6.8: EtherProv tracking & mitigation flow overview

trieved from the provenance database, and consequently stored in the blockchain’s state, such as a dictionary/mapping, to be queried by the mitigation component in real time. Fig. 6.8 shows the tracking and mitigation deployment steps, which we next discuss in detail. When an unhandled security issue is detected, (1) the required contracts are analyzed and (2) the exact compromised paths’ encodings are extracted. (3) The compromised paths’ encodings are added to the PathAccumulator that monitors all participating contracts. (4) When a future call to an instrumented contract is performed, before the transaction is committed, the exact traversed CFG path’s encoding is looked for in the PathAccumulator’s undesired paths. If the path is identified as compromised, (5) PathAccumulator initiates a mitigation strategy, e.g, reverts the transaction. In Section 6.5.4 we provide an example of mitigating undesired paths by reverting the transaction.

6.2 Comparison to related work

In EtherProv, the contracts’ source code (static provenance) is collected. It is further processed to extract the contracts’ full CFG on which the static analysis is

performed. This processed static provenance information helps detect known security issues. The static analysis approaches that are described in the related work (Section 3.2.1) operate on contract bytecode, except for Solc-verify and Slither, that work on high-level Solidity source code. EtherProv extends the approach presented by Slither [66] by transforming the CFG to Datalog relations, which can be queried using Datalog queries. While Vandal and Securify enable querying Datalog relations that contain contract bytecode, EtherProv enables querying Datalog relations that contain contracts' Solidity language constructs. The provided benefits of using the approach used in EtherProv over the approach used by Vandal and Securify include (1) precise analysis, since EtherProv extracts the exact CFG of a contract code, which is challenging to extract from bytecode, and (2) analysis efficiency, i.e., the data structures used in the Solidity source code, e.g., composite structures, arrays, and mappings logically group detached storage address to more coherent structures that are more challenging to deduce from detached bytecode addresses. Further, developers are more familiar with Solidity code constructs and their high-level execution flow; hence, using these constructs in the logic-driven Datalog queries can alleviate the cognitive burden that can be incurred by using bytecode constructs, which can result in less intuitive operations and execution flow analysis.

Subsequently, when a transaction is executed, the execution path's encoding is accumulated during execution (dynamic provenance). The dynamic provenance can be analyzed offline and online (by the contracts). To enable offline future analysis, the dynamic provenance encoding is compressed and stored on-chain. As the dynamic provenance is available during execution, it can be analyzed online by the contract, which can change its execution accordingly. In the the dynamic analysis approaches (described in the related work, Section 3.2.2) SODA and EVM* are complementary to EtherProv as they rely on the modification of a full node to collect run-time dynamic execution data and search for attacks online. In contrast to EtherProv, these

approaches operate on the contract bytecode and use stack-based pattern analysis, which can be less efficient and enable detecting fewer attacks. In contrast to SODA, EVM* enables an interrupting strategy when an attack is detected.

We next discuss how the EtherProv approach differs in comparison to the above dynamic analysis approaches: (1) EtherProv uses logic-driven analysis to detect contract vulnerabilities and attacks, which is more flexible than writing domain specific, hard coded programs for each attack detection; (2) EtherProv enables the collection and analysis of comparable executed provenance granularity, including the executed path trace, with considerably less instrumentation and without the modification of the full node; (3) EtherProv enables a contract’s online reinforcement by instrumenting the contract’s bytecode and not the full node; (4) EtherProv enables analyzing security issues across multiple contract functions in the same contract and across contracts; (5) in addition to attacks detection, EtherProv provides a framework for traceability and forensic analysis across transactions. In comparison to the transaction-centric security analysis approaches (in related work, Section 3.2.3), EtherProv enables detecting all three attack variations presented by Sereum using static analysis. In addition, EtherProv is able to detect various attacks across functions, contracts, and on contract creation. TXSpector and HORUS are perhaps the systems most related to EtherProv. Interestingly, TXSpector specifies that it could not collect all execution traces due to the huge amount of storage required. EtherProv enables alleviating this barrier due to the collection of a compressed encoding of the CFG exact execution path that consists of no more than a few integers per execution trace; subsequently, EtherProv enables detecting vulnerabilities that are not detectable by HORUS and TXSpector, e.g., restricted transfer, which requires to reason about all possible paths, cannot be efficiently detected by analyzing only executed paths. This is due to the fact that EtherProv collects static analysis as well as dynamic analysis data. The static analysis data that are extracted from

the Solidity source code enable analyzing all possible paths. This capability is not possible using transaction execution trace analysis exclusively, as in TXSpector and HORUS, where only paths that have been executed are available for analysis; in addition to vulnerabilities and attacks detection, EtherProv supports online execution reinforcement; Finally, EtherProv operates on Solidity source code CFG and not on the contract bytecode, which provides more accurate CFG extraction and includes added information that is lost in compilation. For example, value types are required to detect the integer underflow/overflow vulnerability; further, the logical data structures used in the Solidity source code, e.g., composite structures, arrays, and mappings, help to logically group detached storage address to more coherent structures that better rationalize about the program’s overall logic. These are lost in the compilation process, which makes the analysis of the program considerably more cumbersome and limited. Preserving this information enables for more efficient forensic analysis.

The large-scale analysis of contracts by Perez and Livshits [109], where only 2% of vulnerable contracts are exploitable, implies that contracts deployed with vulnerabilities may take time to exploit. When an unhandled vulnerability is discovered, EtherProv can help ensure that deployed contracts containing this vulnerability are protected in a timely manner.

Table. 6.3 summarizes the comparison of the related work to EtherProv using the following criteria:

- Provenance extraction type - the type of the extracted provenance.
- Vulnerability detection mode - the mode in which vulnerabilities are detected, i.e., online (during contract execution) or offline.
- Online execution reinforcement - is online enforcement of execution supported.
- Code construct - the code constructs at which the data are collected and ana-

lyzed, i.e., EVM bytecode or Solidity source code.

- Forensic execution path analysis support - if execution path provenance is collected, can it be analyzed on demand.
- Instrumentation - if instrumentation is involved, what is instrumented.
- Audit tracking and analysis - if any type of execution provenance is stored, can it be analyzed.

6.3 Enabling dynamic analysis with higher degrees of soundness and completeness through Solidity source code static analysis data

The static analysis approach is characterized by a higher degree of completeness (the ability of the analysis to cover all possible program behaviors) but with a lower soundness (providing correct and accurate results) due to over approximation. On the other hand, the dynamic analysis approach is characterized by a higher degree of soundness but a lower degree of completeness due to the path explosion problem. EtherProv extracts contract execution CFG paths that are composed of Solidity source code, which are dynamic data that are enriched by static data. This in turn enables enriched dynamic analysis capabilities that enable more storage efficient dynamic data, which in turn can enable dynamic analysis with a higher degree of completeness. Extracting an exact and accurate contract execution CFG path enables a higher degree of soundness.

Enabling dynamic analysis with higher degrees of soundness CFG paths that are extracted from bytecode can incur imprecise executed CFG paths. In contrast, EtherProv's Solidity source code static analysis data enable to extract accurate

dynamic executed CFG paths. Such accuracy is essential to enable a reliable mitigation strategy in real time in order to avoid unintended losses. Further, performing dynamic analysis on bytecode requires to handle raw addresses affiliation to the program logic, e.g., function names encoding and raw memory/storage addresses. In contrast, EtherProv’s use of Solidity high level constructs, e.g., composite structures, arrays, mappings, unencoded variable names, and unencoded function names enable to logically group storage addresses and more efficiently reason about the overall executed path’s logic when performing manual analysis on the dynamic data.

Enabling dynamic analysis with a higher degree of completeness dynamic analysis capabilities are bound by the executed paths, which are expensive to store. EtherProv utilizes efficient path profiling on the extracted Solidity CFG that enables encoding the contract’s executed path to only a few integers that are efficient to store, compared to full traces. Further, at the end of a transaction, EtherProv employs static analysis data to determine which contracts’ states were read/written by the contract for efficient extraction of only relevant execution data.

6.4 Blockchain smart contract security issues

Over the years, several types of smart contract vulnerabilities were discovered. In the following, we will elaborate on a few of these vulnerabilities.

Unchecked Low-Level Calls In 2016 an exception handling bug, which was discovered in the popular contract game “King of the Ether Throne” resulted in an unhandled low-level call failure, which led to the creator of the contract publicly requesting users not to send Ether to it [8]. The issue was caused by the failure to check the return value of a low-level call that indicated whether the instruction had completed successfully or not. Such issues can be prevented by checking that any

low-level call is followed by a condition code that is reliant on the return value of the low-level call.

Controlled Delegatecall The Controlled Delegatecall vulnerability arises when a contract allows external contracts to call into it using the delegatecall function. In a delegate call, the code of the called contract is executed in the context of the calling contract. This means that the called contract can access the calling contract’s storage and execute code on its behalf. If the called contract is malicious, it can abuse this access to modify or steal the calling contract’s assets. In 2018 the the Controlled Delegatecall issue was discovered in the BeautyChain contract, which resulted in \$290,000 worth of Ether [29]. The BeautyChain contract allowed users to submit beauty contest entries, and the contract owner would select the winner based on community votes. The contract used delegatecall to execute external voting contracts on behalf of the users. The attacker created a malicious voting contract, which allowed them to withdraw all the funds from the BeautyChain contract. Such issues can be prevented by identifying and restricting the use of a delegatecall.

Liquid Ether Ethereum contracts enable sending and receiving Ether. The Liquid Ether vulnerability permanently locks funds in the contract. Parity Wallet bug [14] is an example of this type of vulnerability, where in 2017, the removal of a library from the Ethereum blockchain, which was used to exclusively send Ether to other contracts, by a referencing contract, caused an entrapment of \$160M worth of Ether [14].

Re-Entrancy While the Re-Entrancy vulnerability can occur in many forms, it typically requires a contract to “call” another contract or external function multiple times before its previous invocations were finalized, i.e., “re-enter” a function. Re-Entrancy vulnerabilities can occur across multiple functions and multiple contracts. This can cause severe damages, including fully draining funds from vulnerable con-

tracts. An infamous incident of the Re-Entrancy vulnerability occurred in 2016, where a vulnerable DAO contract was exploited, resulting in the stealing of \$60M worth of Ether [13]. The contract included a function call, which sent Ether to the recipient. The function determined the amount to be sent by inspecting a storage variable, which was updated according to the sent amount *after the function call*. The attacker re-invoked the function multiple times, i.e., sending ether, before its dependent storage variable could be updated.

Restricted Writes Ethereum contracts can be accessed publicly; hence, ensuring that only authorised users can modify the contract is essential. Failing to ensure this can enable unrestricted users to change the contract’s behavior and, in some cases, steal its Ether. Such vulnerability was exploited in 2017 when an attacker was able to update the contract’s owner to their own address, resulting in a theft of \$30M [1]. For additional contract security vulnerabilities we refer the interested reader to the list of contract security vulnerabilities addressable by the Slither static analysis framework [126] and by Securify [132].

6.5 Security Evaluation

6.5.1 Implementation

EtherProv was implemented using the Python language with Ganache [4] as an Ethereum blockchain for the deployment and execution of contracts. Slither [66] was used as a third-party Solidity source code static analysis tool and Souffle [80] as the Datalog query engine to run user-defined logic-driven analysis queries against the EtherProv’s provenance database.

6.5.2 Detecting known smart contract vulnerabilities

EtherProv is capable of addressing a wide range of contract security vulnerabilities such as the ones discussed in Section 6.4. In the following, we show how EtherProv addresses the Liquid Ether, Re-Entrancy, and Restricted Writes contract security vulnerabilities.

6.5.2.1 Liquid Ether

EtherProv verifies that the *Liquid Ether* security issue does not occur by checking if a contract can send Ether by either (1) suicide/self destruct function, or (2) a call function with a positive number.

```
1 liquid_ether_compliance(node_id) :- node_with_liquefiable_function_calls(node_id).
2 liquid_ether_compliance(node_id) :- node_with_call_value_not_0(node_id).
3 liquid_ether_compliance(node_id) :- node_with_call_value_dependent_on_sender(node_id).
4
5 node_with_liquefiable_function_calls(node_id) :-
6   non_contract_function_call(node_id, "suicide(address)").
7 node_with_liquefiable_function_calls(node_id) :-
8   non_contract_function_call(node_id,
9     "selfdestruct(address)").
10
11 node_with_call_value_not_0(node_id) :-
12   node(node_id, call_value),
13   node_variable(call_value, name, "uint256", "True"),
14   name != "0".
15
16 node_with_call_value_dependent_on_sender(node_id) :-
17   node(node_id, call_value),
18   node_variable(call_value, _, "uint256", "False"),
19   variable_may_depend_on(call_value, "Client#msg.value").
```

Listing 6.1: Verifying Liquid Ether compliance (EtherProv Datalog rules)

Listing 6.1 shows EtherProv’s corresponding Datalog rules. Overall, a contract does not contain the *Liquid Ether* vulnerability only if: (1) it contains a function that destroys the contract and sends the remaining funds to a specified address (rows 5-7). (2) It contains a call function with a value that is different than 0 (rows 11-14);

(3) It contains a call function with a value, which is dependent on the caller's data (rows 16-19). We next discuss the rules in detail.

Rows 1-3 contain the main rule `liquid_ether_compliance`, which is true only if its sub rules are true. The sub rule `node_with_liquefiable_function_calls` is true if `node_id` references a Solidity reserved function, i.e., `suicide(address)` or `selfdestruct(address)`. These function calls destroy the contract and return all its current Ether to the specified address.

In order for a contract to send Ether in Solidity, a call function is used, which can be any of the following Solidity functions: "call," "send," or "transfer." The amount of Ether to transfer is provided in a parameter. For example, in the following solidity statement: `account_address.call.value(call_value)`, `account_address` is the address of the account/contract to send Ether to, and `call_value` is a variable that contains the amount of Ether to transfer to that address. The sub rule `node_with_call_value_not_0` is true when `node_id` references a call-function that sends Ether to another contract with a parameter referenced by `call_value`, which is static and an integer (parameters 4 and 3 respectively in `node_variable`) with a value other than "0" (parameter 2, `name`, in `node_variable`).

The sub rule `node_with_call_value_dependent_on_sender` is true if `call_value` is non-static and an integer (parameters 4 and 3 respectively in `node_variable`), and is derived from the transaction's data (`Client#msg.value` in `variable_may_depend_on`).

6.5.2.2 Re-Entrancy

Overall, a contract contains the *Re-Entrancy* vulnerability only if: (1) it contains a call-function, and (2) any of the instructions following the call-function are writes to a storage variable. We next discuss the rules in detail.

```
1 no_writes_after_calls_violation(node_id, followed_by_node_id, lvalue_node_variable_id) :-  
2   node(node_id, "LowLevelCall"),
```

```

3  node_may_be_followed_by(node_id, followed_by_node_id),
4  nodes_with_storage_writes(followed_by_node_id, lvalue_node_variable_id).
5
6 nodes_with_storage_writes(node_id, lvalue_node_variable_id) :-
7  node_operation_with_l_value(node_id, lvalue_node_variable_id),
8  node(node_id, "Assignment"),
9  node_variable(lvalue_node_variable_id, "NULL", "True").
10 nodes_with_storage_writes(node_id, lvalue_node_variable_id) :-
11  node_operation_with_l_value(node_id, lvalue_node_variable_id),
12  node(node_id, "Assignment"),
13  node_variable(lvalue_node_variable_id, points_to, _),
14  points_to != "NULL",
15  node_variable(points_to, "NULL", "True").

```

Listing 6.2: Verifying no writes after calls compliance (EtherProv Datalog rules)

Listing 6.2 shows EtherProv’s Datalog rules. Rows 1-4 contain the main rule `no_writes_after_calls_violation`, which is true only if the node `node_id` is a call-function (identified in the schema as "LowLevelCall"), and is followed by a node `followed_by_node_id` that contains an lvalue storage variable (`lvalue_node_variable_id` in `nodes_with_storage_writes`).

`nodes_with_storage_writes` is checked twice with different rules (rows 6-10 and 11-17). It is true if `node_id` references a node that contains `lvalue_node_variable_id`, which references either (1) a storage lvalue (the variable to the left of an assignment) or (2) a temporary variable pointing to a storage lvalue. Its two sub-queries make use of `node_variable(variable_id, points_to, is_storage)`. `variable_id` can reference a non-temporary lvalue or a temporary lvalue, which is a pointer to the real lvalue. In the case that lvalue is a temporary variable, the `points_to` field contains a variable id of the the real lvalue, otherwise it contains `NULL`. The `is_storage` field specifies if the variable is a storage variable ("`True`") or not ("`False`").

6.5.2.3 Restricted Writes

Overall, a contract contains the *Restricted Writes* vulnerability only if: (1) the contract contains a path, which is not a result of a branch, which enables changing a state by any user (i.e., independent of sender), or (2) the contract contains at least one path that is a result of a branch, where the branching condition is independent of the sender, and leads to a storage write.

```
1 restricted_writes_violation(batch_id, root_node_id) :-
2   path_no_branch_with_storage_write_independent_of_sender(batch_id, root_node_id, _, _).
3
4 restricted_writes_violation(batch_id, root_node_id) :-
5   path_yes_branch_with_storage_write_independent_of_sender(batch_id, root_node_id, _),
6   !path_yes_branch_dependent_of_sender(batch_id, root_node_id, _, _).
```

Listing 6.3: `restricted_writes_violation` (EtherProv Datalog rules)

Listing 6.3 shows EtherProv’s Datalog rules. *restricted_writes_violation* is true if (1) *path_no_branch_with_storage_write_independent_of_sender* is true (line 2), i.e., there is a path that is without branches, independent of the sender, and leads to a write; or alternatively, (2) *path_yes_branch_with_storage_write_independent_of_sender* is true (line 5), i.e., there is a path that contains a branch, which is independent of the sender, that leads to a write. If it exists, *path_yes_branch_dependent_of_sender* should not contain its root *root_node_id*. *path_yes_branch_dependent_of_sender* is false (line 6) when the path starting at root *root_node_id* does not contain a branch that is dependant of the sender.

6.5.2.4 Detecting violations across contracts

Existing static analysis tools can analyze only single contracts. Yet, some security issues span over multiple interacting contracts.

```
1 contract Client {
2   LockManager _lm = LockManager();
```

```

3  uint _balance = 1000000;
4  function transferFundsOnce(address dest_address) {
5      if (!_lm.isLocked()) {
6          if (_balance > 100) {
7              _balance = _balance - 100;
8              dest_address.call.value(100)();
9          }
10         _lm.lock();
11     }
12 }
13}
14contract LockManager {
15     bool _locked = false;
16     constructor() public {}
17     function isLocked() returns (bool) {return _locked;}
18     function lock() {_lock = true;}
19}

```

Listing 6.4: Verifying no writes after calls across multiple contracts compliance

Listing 6.4 provides an example of a *Re-Entrancy* vulnerability, which spans multiple contracts. The Client contract (lines 1-13) uses the LockManager contract (lines 14-19). The Client’s function `transferFundsOnce` receives an address to send 100 Ether to. If the lock is set to `false` (line 5), the code continues to send 100 Ether to the `dest_address` address (line 8) and sets the lock to `true` (line 10). Since the lock remains locked, future calls to the function do not send Ether. An attacker can drain Ether from this contract if they are the first caller to the `transferFundsOnce` function and the provided `dest_address` is an address of a contract with a payable fallback function, which contains a call to the `transferFundsOnce` function with its own address. When the function is first invoked, the lock’s state is `false`, and the Ether is sent to the attacker’s contract (line 10), where the fallback function is invoked, which in turn calls the `transferFundsOnce`. Since the lock’s state is not changed yet, an additional 100 Ether are sent to the attacker’s contract with the repeated invocation of the fallback function.

We analyzed the code shown in Listing 6.4 in Slither [66]. Slither was not able to

detect this vulnerability, while EtherProv’s analysis that spans multiple contracts, detected it successfully.

6.5.3 Analyzing new security threats in deployed contracts

EtherProv enables tracing a deployed contract’s historical execution flow, through the source code instrumentation. The emitted encoded path can be retrieved from the blockchain and decoded to the full execution flow without incurring the cost of using an external sandbox environment or maintaining storage state snapshots. The execution flow can be provided as either the ordered source code statements, or as an ordered path’s source code locations, which can be embedded in a debugging tool. Further, EtherProv extracts various dynamic data related to the contract call execution, including its storage state changes, which can facilitate dynamic analysis across contract calls and transactions.

Scenario A Bank’s operations are managed by the Bank contract, while each user’s bank account’s operations are managed by the Client contract. Both are created and deployed to the blockchain. The user’s Client contract is used to send/receive funds to/from other Client contracts. When a user wants to send funds to a designated Client contract, they issue the request to their own Client contract with the amount and address of the designated Client contract’s address. The user’s Client contract queries the Bank contract for the fee, sends it to the Bank contract, and sends the funds to the designated Client contract.

```
1 function _sendAmount(uint amount, Client toClient) {
2   uint fee = _bank.getCurrentFeeContract(_balance, _accountType);
3   uint newBalance = _balance - amount - fee;
4   if (newBalance >= 0) {
5     toClient.addAmountContract(amount);
6     _bank.depositFeeContract(fee);
7     _balance = newBalance;
8   }
```

```
9}
```

Listing 6.5: Solidity Client contract’s internal `_sendAmount` function

This process is implemented in the Client contract’s `_sendAmount` function (Listing 6.5).

```
1 function getCurrentFeeContract(uint balance, string calldata accountType) external
2 returns (uint) {
3     uint sendFee = 0;
4     uint underMinBalanceFee = 0;
5     bool isPreferred =
6     keccak256(abi.encodePacked(accountType)) == keccak256(abi.encodePacked("preferred"));
7     if (isPreferred) {
8         sendFee = _preferredAccSendFee;
9         if (balance < _preferredAccMinBalance) {
10            underMinBalanceFee = _preferredAccUnderMinBalanceFee;
11        }
12    } else { //"accountType == premium"
13        sendFee = _premiumAccSendFee;
14        if (balance < _premiumAccMinBalance) {
15            underMinBalanceFee = _premiumAccUnderMinBalanceFee;
16        }
17    }
18    return sendFee + underMinBalanceFee;
19}
```

Listing 6.6: Solidity Bank contract’s `getCurrentFeeContract`

Listing 6.6 provides the details of the `getCurrentFeeContract` function, which determines the client account’s fee.

Security concern 1 The bank has created 2 Client contracts: `client1` and `client2` for `user1` and `user2` respectively. Both users should have a “preferred” account type. `User1` deposits 2000 credits into their `Client1` contract, and then issues a request to its `Client1` contract to transfer 100 funds to the `Client2` contract; however, `User1` sees that the fee deducted from their Client contract’s balance is larger than it should be.

state_parameter_id	current_value	prev_value
Client#_balance	1884	2000

Table 6.1: contract_call_changed_states query results

Security concern 1 analysis EtherProv provides sufficient provenance data to dynamically examine data flow across these deployed contracts.

Function `getCurrentFeeContract(_balance, _accountType)` (Line 2) is called on the Bank’s contract to determine the bank’s fee, which is dependent on the `accountType`, which can be “preferred” or “premium”. The fee is composed of a fixed fee for each sending of funds and an additional charge if the Client’s balance is below a certain threshold. In order to confirm the fee amount that was charged, EtherProv enables analyzing the historical fee details that were associated with the client’s transaction `9f0efee0...`

```

1 contract_call_changed_states(state_id, current_value, prev_value) :-
2   contract_call_written_states(_, "9f0efee0...",
3   state_id, current_value, prev_written_state_id),
4   contract_call_written_states(prev_written_state_id, _, _, prev_value, _).
```

Listing 6.7: Contract call changed states (EtherProv query)

Listing 6.7 provides the EtherProv’s query that extracts the value of all state changes resulting from contract call ("`9f0efee0...`"). For each record in `contract_call_written_states`, the first field contains the contract parameter’s unique identifier, the second field contains the parameter’s current value (after the specified contract call), and the third field contains the parameter’s previous value (as it was before it was changed by any previous transaction). The query output is shown in Table 6.1. For the `_balance` parameter, $current_value = 1884$ and $prev_value = 2000$. From this the fee can be computed as $2000 - 1884 - 100 = 16$, which is indeed different from the Bank contract’s fee related to the “preferred” account.

```

1 decoded_path(path_id, edge_id, from_node_id, to_node_id, type, expression) :-
2   dynamic_path("9f0efee0...", path_id, _, _),
```

row	path_id	edge_id	from_node_id	to_node_id	type	expression
1	6	252	Client#sendAmount...	Client#sendAmount	EXPR	sendAmount(amount,toClientAddress)
2	6	231	Client#sendAmount	Client#_sendAmount	EXPR	_sendAmount(amount,toClient)
3	6	236	Client#_sendAmount	Bank#getCurrentFee...	NEW	fee = _bank.getCurrentFee...
4	6	199	Bank#getCurrentFee...	Bank#getCurrentFee...	NEW	sendFee = 0
5	6	200	Bank#getCurrentFee...	Bank#getCurrentFee...	NEW	underMinBalanceFee = 0
6	6	201	Bank#getCurrentFee...	Bank#getCurrentFee...	NEW	isPreferred = keccak256(bytes)(ab...
7	6	202	Bank#getCurrentFee...	Bank#getCurrentFee...	IF	isPreferred
8	6	203	Bank#getCurrentFee...	Bank#getCurrentFee...	EXPR	sendFee = _premiumAccSendFee
9	6	204	Bank#getCurrentFee...	Bank#getCurrentFee...	IF	balance < _premiumAccMinBalance
10	6	205	Bank#getCurrentFee...	Bank#getCurrentFee...	EXPR	underMinBalanceFee = _premiumAccUnd...

Figure 6.9: Sampled results of decoded path from EtherProv query

```

3 static_path(path_id, edge_id),
4 static_edge(edge_id, from_node_id, to_node_id),
5 static_node(from_node_id, _, type, expression).

```

Listing 6.8: Decoded path (EtherProv query)

To enable a root-cause analysis of the difference, the query in Listing 6.8 is executed to extract the execution flow provenance of the contract call involving the deployed Client and Bank contracts in the relevant transaction.

A sample of the query results are shown in Fig. 6.9. Rows 7-8 show the execution flow of the function `getCurrentFeeContract` (Listing 6.6), where the fee is determined. It shows that the branch regarding the “premium” account was taken instead of the branch regarding the “preferred” account. The analyst continues to query the Client’s `accountType` state value and discovers it was entered with a typo as the “prefered” account type.

Security concern 2 The Client contract contains a bug in `_sendAmount` (Listing 6.5 row 3). The statement `uint newBalance = _balance - amount - fee;` contains an assignment to a variable of type `uint`. A negative number assignment to a `uint` results in an integer underflow, which consequently may result in a considerable large number. For example, a user sends funds to a Client where the sum of the fee and the amount are larger than the balance. Following the transaction, the user obtains a large balance amount. Consequently, all transactions that involve this large balance contribute to the magnitude of the issue.

Security concern 2 analysis This integer underflow becomes obvious when querying specific paths, e.g., paths that contain a call to `_sendAmount` with focus on states of type `uint` that are known to be strictly decreasing (e.g., `balance` variable when sending funds). If the states' values are increasing between subsequent transactions, an integer underflow is detected.

EtherProv enables detecting tainted data by following the flow of tainted values, i.e., when a contract call operates on tainted values, its output storage states are also considered tainted. In the integer underflow example, the first occurrence of the (tainted) large balance value can be used to query all contract calls that used either this value directly or other values tainted by this value.

```
1 tainted_state_ids(tainted_state_id) :-
2 tainted_state_ids("8093c811...").
3 tainted_state_ids(new_tainted_state_id) :-
4   tainted_state_ids(known_tainted_state_id),
5   state_parameter_read(contract_call_id, _, known_tainted_state_id),
6   state_parameter_written(_, contract_call_id, _, _, new_tainted_state_id).
```

Listing 6.9: Extracting tainted state ids (EtherProv query)

Listing 6.9 provides the recursive query that retrieves all tainted state values, across contract calls, that were computed directly or indirectly by using the initial tainted state value (with ID of "8093c811..."). In order to locate all account keys that used tainted data, the contract calls that used the tainted data need to be retrieved. From these contract calls the calling account key can be retrieved.

```
tainted_contract_call_ids(tainted_call_id) :-
    tainted_state_ids(tainted_state_id),
    state_parameter_read(tainted_call_id, _, tainted_state_id).
```

Listing 6.10: Extracting tainted contract call ids (EtherProv query)

Listing 6.10 provides the query to retrieve the affected contract call IDs (provided by the field `tainted_call_id`).

To explore scalability of our approach, we run the query given in Listing 6.9 with

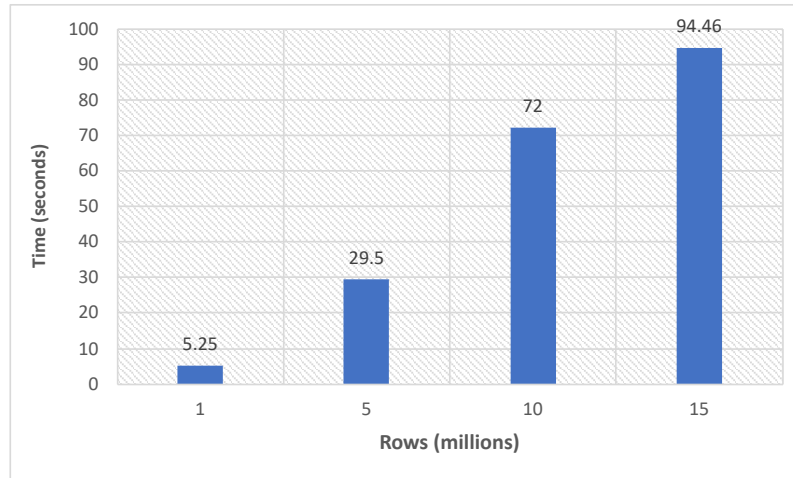


Figure 6.10: Scalability analysis of Listing 6.9 query

large amounts of contract call data. As Fig. 6.10 shows, the performance scales in a linear manner.

6.5.4 Mitigating security threats in deployed contracts

Fixing an unhandled security issue or bug in a deployed contract is challenging, as a deployed contract is immutable. In addition, it is essential for the mitigation to be accurate and reliable to prevent further monetary loss.

EtherProv enables an accurate and reliable mitigation of such issues.

```

1 function flush_path_data() {
2   if (_revert_paths[_current_path.path_id] != 0) revert();
3   emit path(_current_path.path_id, _current_path.count);
4   _current_path.is_init = 0;
5 }
6 function update_reverted_path_id(uint[] calldata path_ids, uint size) {
7   for (uint i=0; i<size; i++) _revert_paths[path_ids[i]]=1;
8 }

```

Listing 6.11: PathAccumulator partial Solidity code

Listing 6.11 shows the partial `PathAccumulator` code related to emitting the accumulated executed encoded path in run-time (called from instrumentation) and reverting specified paths if such are detected. The `_revert_paths` dictionary is used

to store all paths that should be reverted in run-time before completing the transaction. Lines 6-8 show the function used by the `PathAccumulator` owner to update the dictionary with those predefined paths. Upon transaction execution, before emitting the current executed path, if `_revert_paths` contains the path (line 2), the transaction is reverted.

We evaluated EtherProv’s mitigation capability on the original “King of the Ether Throne” contract, `KingOfTheEtherThrone.sol` [9]. The contract was instrumented and deployed. We then issued a query that extracted all paths encoding containing a call to the `claimThrone` function, which were then inserted to the `_revert_paths` dictionary. The subsequent contract calls were reverted successfully.

As another example, consider the tainted data security issue presented in the previous section. After detecting the integer underflow, the bank would need to block any further, potentially damaging, Client contracts’ operations such as depositing, sending, or receiving funds, using possible tainted data. Note that the Client contracts also provide benign functionality such as querying current balance, or transaction fee. Such capabilities should not be blocked. In EtherProv, all paths, which contain depositing, sending or receiving of funds should be extracted and inserted to the aforementioned dictionary. This will allow to block potential harmful Client contracts’ functionality, while still enabling benign functionality.

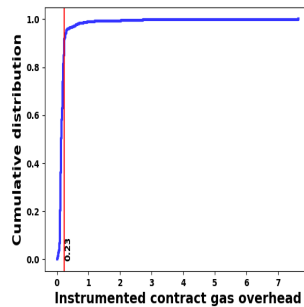


Figure 6.11: Instrumented contracts’ average gas overhead CDF

Contracts count	Avg gas overhead	Avg gas overhead std
1652	18.90%	0.378

Table 6.2: Instrumented contracts statistics

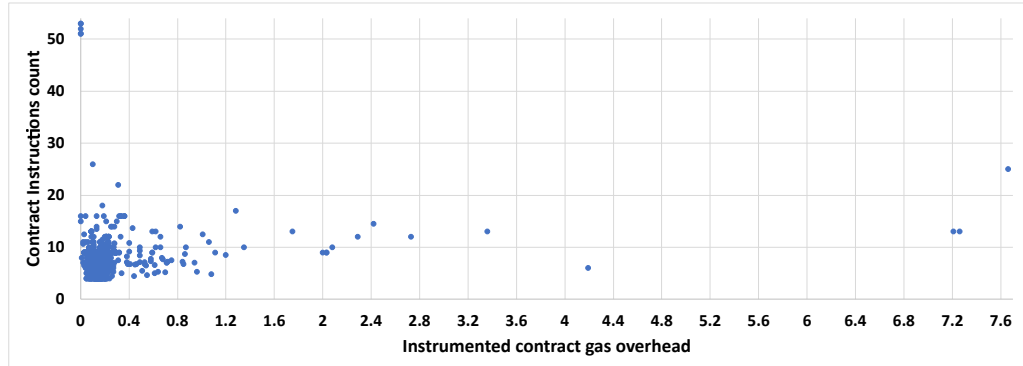


Figure 6.12: Instrumented contracts

6.6 Performance evaluation

In this section we evaluate 1652 instrumented contracts’ execution gas overhead (Table 6.2). The contracts’ Solidity source code were retrieved from etherscan.io [23] and the evaluation was performed on the Ganache platform. For each contract, we identify its external functions and uncalled public functions. Each such function is executed with its instrumented and uninstrumented versions for which we calculate the respective gas overhead. When executing an instrumented contract’s function we extract from the emitted executed path the number of executed instructions, which is the number of executed instructions in the uninstrumented contract. We calculate a contract’s execution gas overhead as $\frac{\sum_{f \in \text{Contract}} (f_{instrumented}^{gas} - f_{uninstrumented}^{gas})}{\sum_{f \in \text{Contract}} (f_{uninstrumented}^{gas})}$, where f is a contract’s executed function.

The contract’s corresponding number of instructions (*instructions_count*) are computed as the average of all executed uninstrumented contract functions’ instructions number. The average contract gas overhead received is 18.9%. In comparison, the study by Wang et al. [134] reported an average run-time overhead of 28.27%. Fig. 6.11 shows that 90% of the contracts (vertical red line) incur a maximum gas overhead of 23%.

The instrumentation of a contract containing loops uses a path compression logic before emitting a loop encoding. Our instrumentation of contracts without loops is lightweight and hence incurs considerably lower gas cost. Fig. 6.12 helps to better understand the varying instrumented gas overheads. We manually analyze the three corner cases in the graph:

- *High instructions count and low instrumented contract gas overhead* (top left of the graph). We analyzed the top 5 contracts with the highest ratio of

$$\frac{\text{instructions_count}}{\text{instrumented_contracts_gas_overhead}}$$

Most of the executed functions' paths in the extracted uninstrumented contracts contained a high number of uninstrumented instructions alongside low instrumentation due to lack of branching or loops, which helps explain the low gas overhead. Specifically, some functions' paths contained 50-60 instructions (e.g., write hard coded lists into a map storage). Since storage write is the most expensive instruction, its contribution to the instrumentation gas overhead is negligible.

- *Low instructions count and low instrumented contract gas overhead* (bottom left of the graph). We examined the top 5 contracts with the lowest instrumented gas overhead and the lowest instruction count. Most of the executed functions' paths in the extracted uninstrumented contracts contained few instructions and low instrumentation (due to lack of branching or loops), which helps explain the low instrumentation gas overhead. Specifically, some function paths contained 5-13 write to storage instructions (most gas consuming), which helps explain the considerably lower gas overhead.

- *Lower instructions count and high instrumented contract gas overhead* (bottom right of the graph). We analyzed the top 5 contracts with the highest ratio of

$$\frac{\text{instrumented_contracts_gas_overhead}}{\text{instructions_count}}$$

Most of the executed functions' paths contained few instructions and high instrumentation due to loops and branching, which helps explain the high instrumentation gas overhead. Specifically, some executed function

paths contained 1-2 storage or memory reads/writes or an emit instruction (all of which consume little gas) and 1-2 branches and a loop, which contribute to higher instrumentation and thus explain the considerably higher instrumentation gas overhead.

We examined some of the instrumented contracts in between the corner cases and found that the ratio of instrumentation to the cost and frequency of uninstrumented instructions correlates with the logic discussed in the corner cases, i.e, more/less costly executed instructions with corresponding less/more branching or loops result in low/high gas overhead respectively. The results show that instrumentation is the most efficient on contracts without loops. However, contracts with loops incur a gas overhead relative to the loops' frequency.

6.7 Complexity evaluation

We evaluate the complexity of the instrumentation process. We start by evaluating each part of the algorithm and conclude with the total complexity.

In our analysis, $V =$ Vertices (or nodes) and $E =$ Edges.

The algorithm is composed of 5 main parts: (1) the efficient path profiling algorithm initially locates all nodes that have no parents (to be pointed to by the “entry” node) and all nodes that have no children (to point to the “exit” node). This is achieved by traversing each node and checking its children/parents with a cost of $O(V + E)$. (2) The efficient path profiling algorithm requires to locate loops in order to reallocate their back-edges. Finding back-edges of loops is done by using a Depth First Search (DFS) with a cost of $O(V + E)$. (3) Assigning unique encoding to each path in the efficient path profiling algorithm is achieved by first finding the reverse topological ordering of the DAG using Kahn's algorithm with a cost of $O(V + E)$. Following, each edge is assigned an increment to be added when traversing the path, which

enabled each path’s unique encoding with a cost of $O(V + E)$. (4) To optimize the number of instrumented edges that contribute to each path encoding, the efficient path profiling algorithm adds an edge from the “exit” node to the “entry” node and computes a minimum spanning tree (MST). However, since we do not consider edges’ weight the calculation can be of a regular spanning tree, which can be computed using Kruskal’s algorithm with a cost of $O(V + E)$. The algorithm continues to instrument only edges that are not on the MST, which results in optimized instrumentation. The instrumentation algorithm of these edges is provided in the work by Ball [33]. In this algorithm, each edge, that is not in the MST, is represented by its “from”/“to” vertices. The algorithm operates on the fact that for each such edge, there is a path in the MST from the “to” vertex until it reaches the “from” vertex. The encoding of this path is computed by summing the increments of each traversed edge in the MST. If such a path includes some traversal of the directed MST in a reverse order, the increment of a reverse traversed edge is negated before adding it to the running sum. The resulting summation is added to the increment value of the original edge (that is not in the MST) and is set as the instrumentation of that original edge. This algorithm can be computed using DFS at the cost of $O(V + E)$. (5) At this stage the original CFG is instrumented with the resulting optimized instrumentation. We next traverse all of the instrumented edges and inject the instrumentation in the source code using heuristics with a cost of $O(V + E)$. Hence, the total complexity of all steps is $O(V + E)$.

	Provenance extraction type	Vulnerability detection mode	Online execution reinforcement	Code construct	Forensic execution path analysis support	Instrumentation	Audit tracking/analysis support
Oyente	Static	Offline	NA	Bytecode	NA	NA	NA
Mythril	Static	Offline	NA	Bytecode	NA	NA	NA
Sole-Verify	Static	Offline	NA	Bytecode	NA	NA	NA
Vandal	Static	Offline	NA	Bytecode	NA	NA	NA
Securify	Static	Offline	NA	Bytecode	NA	NA	NA
Slither	Static	Offline	NA	Solidity	NA	NA	NA
SODA	Dynamic	Online	X	Bytecode	X	Full node	X
EVM*	Dynamic	Online	V	Bytecode	X	Full node	X
Sereum	Dynamic	Online	X	Bytecode	X	Full node	X
ECFChecker	Dynamic	Online	X	Bytecode	X	Full node	X
Horus	Dynamic	Offline	NA	Bytecode	X	Full node	V
TXSpector	Dynamic	Offline	NA	Bytecode	V	Full node	V
EtherProv	Static/Dynamic	Online/Offline	V	Solidity	V	Contract bytecode	V

Table 6.3: Comparing EtherProv to related work

Chapter 7

De-anonymizing Ethereum Blockchain Smart Contracts through Code Attribution

In chapter 6 we provided a framework that can efficiently handle security vulnerabilities and attacks before and after a contract is run. When an account address was determined to be related to an attack, finding the attacker's identity and their related account addresses is crucial to enforce the full scope of their accountability and to determine and remediate all blockchain activities that may have been affected by the attacker. In this chapter we provide an approach to affiliate related account addresses that are used to deploy contracts using stylometry techniques and explore its validity.

Since blockchain technologies may entail significant monetary value to their users, attacks on these platforms are mounting [48,92]. The majority of these attacks are profit driven and leverage the fact that the identity of an adversary is hidden behind the account address, which cannot be linked back to them without out-of-network information. In fact, the pseudo-anonymity of an account address is one of the main

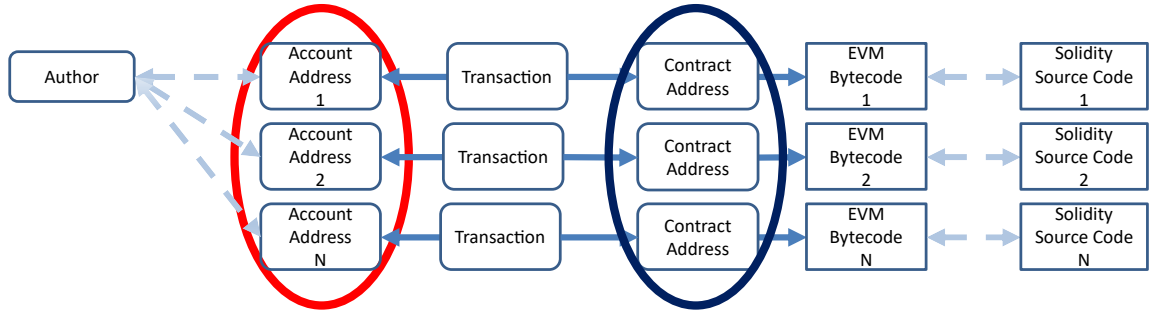


Figure 7.1: Account address affiliation using smart contract code attribution

premises of public blockchain platforms such as Ethereum, which is paramount due to all transactions being publicly available. Hence, users can and are encouraged to generate a new account address per transaction. At this point, the only measure that allows combating the blockchain abuse is to drop transactions that contain suspicious account addresses from malicious users during the transaction validation. However, detecting suspicious account addresses remains challenging.

In this work, we propose an approach to link together addresses that are used to deploy contracts, which are produced by the same author. Fig. 7.1 helps to present our motivation. An author can create multiple account addresses (red ellipsis) that are not directly linked to them. Each account address can be used to deploy a contract's EVM bytecode to the Ethereum blockchain, which is assigned a contract address. A transaction is created to link the deployed contract's address to the deploying account address. The contract's Solidity source code is not deployed to the blockchain and hence cannot be directly linked to the deployed contract. By enabling deployed contract addresses produced by the same author (dark blue ellipsis) to be linked, their deploying account addresses can also be linked. Such affiliation information can be used to discriminate against an account address, which was linked to a suspicious group of addresses. Further, if an out-of-network information is available on few account addresses, this can lead to the de-anonymization of all their linked account addresses.

Previous research on de-anonymization explored the affiliation of bitcoin addresses

by using out-of-network information such as IP addresses [87, 119], geo-locations [54], inner network information using graph analysis [116], and bitcoin address classification techniques [44, 99]. We take an alternative approach and leverage stylometry techniques, widely used in the social sciences for attribution of literary texts to their corresponding authors. The assumption underlying stylometric analysis is the existence of a distinctive writing style, unique to an author and easily distinguishable from others. Within this analysis, we rely on an insight from code authorship attribution techniques that allow the identification of a code’s developer based on the unique characteristics that describe the developer’s coding style. This style can be expressed through layout (e.g., indentations, white spaces), lexical (e.g., function lengths, variable names) and syntactic (e.g., control flow structure, AST depth) levels. Research on authorship attribution demonstrates the effectiveness of these types of features on accurate attribution of source code [43, 49].

In our approach, we explore the coding style of Ethereum contracts’ developers at the level of the Solidity source code and its corresponding bytecode, which is deployed on the Ethereum blockchain. Attribution of blockchain contracts presents several challenges. Bytecode, in general, including the Ethereum contract’s bytecode, preserves only few of the stylistic properties of the corresponding source code; thus, it is desirable to apply attribution analysis on the original source code. Since a deployed contract does not preserve its source code, we obtain the contract’s Solidity source code from `etherscan.io` [23], which enables a contract’s author to upload the Solidity source code to their platform and link it to the deployed contract address on the blockchain. While this type of manual pairing is not a mandatory process, it can enhance the credibility of the contract as it improves transparency. The Ethereum blockchain’s environmental constraints impose several restrictions on contract code compared to ”traditional” non-blockchain, including reduced code size, a preference for more gas-effective instructions, concise data manipulation, a small pool of re-

served keywords, and repeated coding patterns use. These restrictions result in compact and simple code, which may make it challenging to extract unique features that characterize a developer’s coding style. We discuss these differences in more detail in Section 7.1. Under these constraints code reuse can significantly impact contract code authorship attribution through unnecessary attribution bias. However, by removing code duplication, meaningful and unique features can be extracted to enable efficient attribution of contract authors using traditional attribution methods.

To address these concerns and reduce code duplication, we introduce multiple data refinement heuristics, which are based on the contracts’ Solidity source codes, and explore feature selection techniques to extract the most effective features. In the analysis, we focus on two commonly used approaches in code attribution: attribution based on n-grams and attribution based on a state-of-the-art feature set proposed by Caliskan et al. [43]. We validate our approach on our extensive dataset with verified real-world Ethereum contract data. Our experimental results show that it is feasible to attribute Ethereum contracts to their corresponding deployers with a small subset of features. We were able to attribute Solidity source code with 91% accuracy, 87% recall, and 84% precision, while using less than 1% of the total features. In attributing contracts represented by EVM bytecode we achieved 80% accuracy, 74% recall, and 71% precision, while using less than 3% of the total features.

We further explore our approach on real-world blockchain abuses such as scam contracts used by adversaries in a Ponzi scheme and other real-world scammers data. We additionally provide an algorithm to extract *distinctly contributing* features from a dataset as a whole and from specific authors. We extract and explore such features in our dataset and in the Ponzi scheme dataset.

7.1 Difference between smart contract code and “traditional” non-blockchain code

Ethereum smart contracts’ high level Solidity source code and EVM bytecode representations have different characteristics from “traditional” non-blockchain high level source code and bytecode. This is due to the specific environmental constraints in which the Ethereum contract code is required to operate on. Since Ethereum contracts are required to participate in the consensus protocol all Ethereum nodes need to share the EVM bytecode, which consumes network bandwidth, and require to run the contract code to achieve the transformed blockchain state that results from running their logic, which consumes computation resources. For these reasons, the code is required to be lightweight in size and not be excessively demanding in terms of computation. In order to help enforce these conditions the Ethereum execution environment imposes several restrictions: (1) reduced code size - each contract execution is allocated a fixed amount of gas, where each instruction consumes a portion of that gas. This requirement results in code with a limited amount of instructions. Further, the maximum size of a contract bytecode is restricted to (24 KB) as contracts that are too large can put a strain on the blockchain’s resources and slow down overall transaction processing [42]; Consequently, developers try to reduce the bytecode size by delegating some functionality to libraries, which in turn results in extensive contract code reuse; (2) preference for using specific instructions over others - each instruction consumes a different amount of gas. Some of the most gas consuming instructions, in a descending order of the amount of gas consumption are: creating a new smart contract within a smart contract execution, calling an external contract, setting a blockchain storage state with a value, reading a blockchain storage state, and appending a log record. In comparison, some of the least gas consuming instructions are: jump instruction (e.g., if condition), arithmetic operations, and memory oper-

ations (as opposed to storage operations). This requirement results in a code with a clear preference for using more gas effective instructions over gas intensive ones; (3) concise data manipulation instead of data management - the above restrictions imposes gearing contract logic towards data computation/manipulation (cheaper) as opposed to data management (expensive). Specifically, since the contract is bound by gas usage, it is restricted to performing fixed and concise computations, which should not be dependent on other data values, e.g., array or loops iterations that are bound by a dynamic value; (4) small pool of reserved keywords - as contract code is designed to support concise operations on limited data the programming possibilities are intentionally limited, which does not require a rich set of keywords to encapsulate richer and divers capabilities are is non-blockchain languages. (5) repeated coding patterns use - after deploying a contract it is immutable and cannot change. As a result, it should be secure and correct. This limitation encourages the use of best practice design patterns to reduce such issues.

The above restrictions result in contract code that is compact and geared towards simple fixed and concise operations (as opposed to data management) with a limited number of reserved keywords with a bias toward more frequent less gas consuming instructions. In addition, the use of similar coding patterns increased code similarity of contracts. As a result, it may be more challenging to extract unique features that can uniquely characterize a developer's coding style from contract code compared to extracting unique features from "traditional" non-blockchain code. In addition, since only the contract's bytecode is required to be maintained on the blockchain, Solidity source codes can be difficult to obtain. This in turn may present code attribution challenges that are unique to blockchain related programs. In this chapter we show that due to the above contract code constraints code reuse can be detrimental to the effectiveness of contract code authorship attribution. By removing code duplication using different heuristics we show that it is possible to extract meaningful and unique

features that enable to efficiently attribute contract authors to their deployed code using traditional attribution methods.

7.2 Smart contract attribution

Since pseudo-anonymity is a key feature of a public blockchain technology, the Ethereum platform (like other public blockchain implementations) is designed to maintain no personal identifiable information. The only source of data that can be directly linked to a blockchain user’s account address, beyond the cryptographic keys, is transaction information. Each transaction contains the addresses of the transaction’s issuer and receiver. In the case where an account holder executes a transaction in order to deploy a contract, the transaction will contain the account address and the deployed contract’s code/address. In our approach, we explore the feasibility of attributing deployed contracts’ source codes and bytecodes to their deployers’ account addresses based on the coding style. We turn our attention to code attribution research that showed the effectiveness of attribution techniques in their ability to identify an author of a given code based on extracted coding style. We examine the two commonly used approaches in code attribution: attribution based on n-grams and attribution based on a customized feature set proposed by Caliskan et al. [43].

The overview of the process is shown in Figure 7.2. At the high-level it comprises three main parts: *Feature selection*, *Heuristic refinement*, and *Authorship attribution*. For each Ethereum contract, we obtain both source code and bytecode. The *Feature selection* module extracts features from the contract’s Solidity source code and the EVM bytecode and, by employing classification, determines the best performing features. Due to the potential scale of the available features, at this stage we also attempt to reduce the dimensionality of the feature vectors by incrementally adding

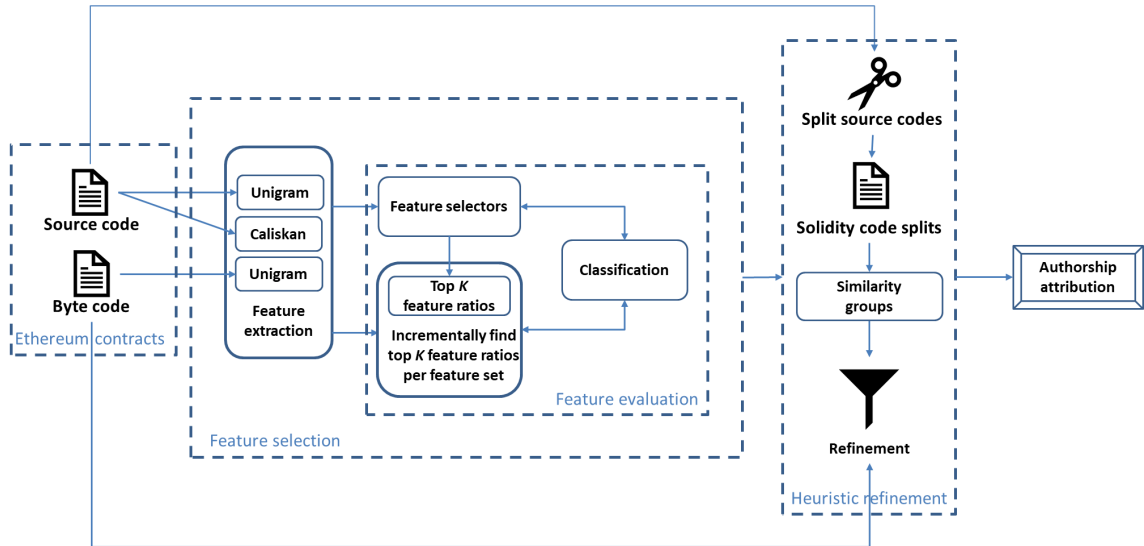


Figure 7.2: High-level flow of Ethereum smart contracts authorship attribution

top K features, until an acceptable classification accuracy is achieved. The top K features' ratios (from the overall number of features) are then passed to the *Heuristic refinement* stage. Along with the features, at this stage we also attempt to determine the best performing classifier.

Authorship attribution assumes that contracts are fully written by individual developers, yet in reality software developers reuse code, third-party libraries, and tools that introduce new stylistic features. To reduce the bias and remove duplicate code pieces, the *Heuristic refinement* module splits each contract's source code into individual components and categorizes similar splits between contracts into corresponding similarity groups. The Solidity source code splits, similarity groups, and bytecodes are used as input to each refinement heuristic. We discuss these in more detail in Section 7.2.3. During this refinement process, for each heuristic refinement of our dataset, we apply the selected top K features' ratios per feature set. The output of this stage is the best performing classifier and the best feature sets per heuristic refinement. The final step is the authorship attribution of contracts using the selected characteristics. The process is described in more detail in the following sections.

7.2.1 Extracting Ethereum smart contracts

When a contract is deployed to the blockchain, only its bytecode is retained on the chain. To obtain the original source code, reverse engineering techniques can be applied on the bytecode to a limited degree, which is affected by the compilation process into the EVM bytecode, e.g., optimizing the contract’s bytecode for performance may change the contract’s code structure while maintaining the same functionality, human readable variable names are not required by the EVM and are encoded, and the layout information of the source code is removed. This makes the reverse engineering of the exact original source code challenging. Some tools were proposed to reverse engineer a contract’s bytecode to a human readable source code with limited success, e.g., porosity [24], and radare2 [25]. `etherscan.io` [23] is an Ethereum blockchain explorer platform, which provides a range of capabilities that include the retrieval of contracts’ source codes in specific cases, without the use of reverse engineering. It does so by providing an API to upload the source code to `etherscan.io`. The source code is then compiled into bytecode and is compared with the deployed contract’s bytecode. If the two are equal, the contract’s source code is considered verified. This manual pairing is entirely optional and is not reflected in the Ethereum blockchain. In our analysis, we rely exclusively on verified contracts that provide both bytecode and source code representations. We further disassemble the bytecode into opcodes for easier parsing. Since Solidity is the most commonly used programming language for writing Ethereum contracts, we focus our analysis solely on contracts written in Solidity.

7.2.2 Feature selection step

In our analysis, we adopt two feature sets that are widely used in code attribution studies and which are treated as benchmark sets: n-gram features employed by [70, 88, 122, 129], and a feature set derived by Caliskan et al. [43], which is used in the

Code type	Feature set
Source Code	Unigram
	Caliskan
Source Code Splits	Unigram
Bytecode	Unigram (of opcode)

Table 7.1: Feature sets per code type

majority of recent studies [49, 124].

N-grams are derived at the lexical level and thus they primarily represent the layout characteristics of the code. In contrast, the Caliskan et al. [43] feature set includes lexical features (e.g., indentation, white space use, braces placements, statistical distribution of variable lengths, and capitalization), and syntactic features that outline the external structural organization of the code and include features which are derived from an AST, e.g., code length, nesting levels, and branching.

To obtain n-gram features we tokenize both source code and opcode (extracted from the bytecodes) files using space, carriage return, new line, and tab. The Caliskan et al. feature set was originally developed for C and C++ programs; we thus map these features to the corresponding Solidity features as outlined in Appendix C.1. Table 7.1 summarizes the different feature sets that are used by the source code, bytecode, and source code splits code types, which we explore in the following sections.

For classification, we explore the Support Vector Machine (SVM) and Random Forest (RF) algorithms, both widely used in authorship attribution research [30, 81, 82, 120, 123]. SVM computes the support vectors that maximize the margin between classes. It is especially effective on high dimensional vectors, which are extensively used in authorship attribution, in terms of space and computation complexity. On the other hand, SVM is susceptible to overfitting. Random Forest (RF) is an ensemble learning algorithm that makes a good trade-off between accuracy and overfitting by employing multiple Decision Trees (DT), where each DT is trained on a random subset of features. In the following sections we explore a RF classifier (with 100 trees and the *gini* purity measurement [37]) and a SVM classifier (with the radial basis

function kernel (RBF)). We opted for the RBF kernel in the SVM classifier since the dataset is extensive, and the number of unigrams is substantial. The RBF kernel is a suitable option in such cases, as it can effectively handle the high-dimensional feature space and identify non-linear associations between the unigrams and the output labels.

7.2.3 Heuristic refinement step

The tendency of programmers to reuse their own code components and those written by others can negatively affect the code’s classification accuracy [41, 117]. Solidity, like most programming languages, enables the creation of code libraries for common algorithms reuse [11]. These libraries can be created in a local or a remote contract. In the various Solidity contracts’ source codes that we examined, we located only a few remote library calls. Instead, the common approach is to copy necessary code into a contract and modify it as required. We examined the top 10 similarities in our Solidity source codes’ dataset and found that the common components include (in descending order) ERC23 contract interfaces, safe math libraries, ownership contract implementations, ERC20 contract implementations, token recipient interface implementations, and pausable interface implementations. The results show that most similarities are attributed to token standards implementations (e.g., the evolution of token standards from EC20 to EC23 which introduced a new code template to copy and reuse) as well as a generic safe math functionality. This introduces a large amount of duplicate code and increases the similarity between contracts.

To avoid unnecessary attribution bias and reduce the amount of common code in the contracts, we consider several data refinement heuristics. These heuristics depend on the contract’s Solidity source code and provide the code similarity assessment at the component granularity level, such as contracts, libraries, and interfaces. Note that Ethereum supports inheritance and, therefore, a single program’s source code

may include multiple contracts that inherit characteristics from each other.

Before the refinement is applied, each contract's source code is split into its individual components. Similarity between splits is assessed using the Levenshtein distance metric. To reduce the number of pairwise comparisons, each split is compared to another split only if their size differs by at least 10%. Two compared splits are considered similar if their similarity score is 80% or higher. From each similarity group we retain a single split in an arbitrary way and delete all other similar splits according to one of the following heuristics:

1. Considers splits' similarity between authors as well as within an author - if any contract's split was found to be similar, and hence removed, all the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. The remaining splits per contract are merged to produce a single compilable contract source code.
2. Considers splits' similarity between authors as well as within an author - if any contract's split was found to be similar, and hence removed, all the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. Each split is treated as a full contract source code for classification purposes.
3. Considers splits' similarity between authors as well as within an author - only contracts which had all their splits removed (due to being similar to other splits) are deleted. The remaining splits represent full or partial contracts without any similarities. The remaining splits per contract are merged to produce a full or partial contract source code which may be non-compilable due to its partialness. This can prevent the extraction of ASTs that are required for the Caliskan feature extraction. Subsequently, we only extract unigram features. For each merged source code, we retrieve the opcodes corresponding to the original full source code before the splits' removal.

4. Considers splits' similarity between authors as well as within an author - only contracts which had all their splits removed (due to being similar to other splits) are deleted. Each split is treated as a full contract source code for classification purposes.
5. Considers splits' similarity only between authors - if any contract's split was found to be similar, and hence removed, all the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. The remaining splits per contract are merged to produce a single compilable contract source code.
6. Considers splits' similarity only between authors - if any contract's split was found to be similar, and hence removed, all the contract's splits are removed as well. The remaining splits belong to contracts without any similarities. Each split is treated as a full contract source code for classification purposes.
7. Considers splits' similarity only between authors - only contracts that had all their splits removed (due to being similar to other splits) are deleted. The remaining splits represent full or partial contracts without any similarities. The remaining splits per contract are merged to produce a full or partial contract source code which may be non-compilable due to its partialness.
8. Considers splits' similarity only between authors - only contracts that had all their splits removed (due to being similar to other splits) are deleted. The remaining splits represent full or partial contracts without any similarities. The remaining splits per contract are merged to produce a full or partial contract source code which may be non-compilable due to its partialness.

A heuristic refinement running cost is mostly affected by the construction of the splits' similarity groups. This construction includes comparing every two splits of different contracts and grouping all similar splits into their own similarity group, with

an overall cost of $\mathcal{O}(S^2)$, where S is the number of contracts’ splits. Once the splits’ similarity groups are constructed, they can be used by all heuristic refinements. Each heuristic refinement iterates over the contracts’ splits in each similarity group with a total cost of $\mathcal{O}(S)$.

7.3 Evaluation

7.3.1 Data

Since pseudo-anonymity is an important feature of all blockchain technologies, there are no datasets available for research that identify users and their corresponding transactions or contracts (in the case of Ethereum); therefore, there is no “ground truth” data for our experimentation purposes. To ensure a comprehensive evaluation of the proposed approach and explore its feasibility, we constructed a validation dataset with known relations between users and contracts (as represented by the deployed contracts’ account addresses). Generally, it is possible for an individual user to generate multiple transactions under different keys. In this work, we adopt a conservative approach and assume that each user is identified by a single account address, which can be used to deploy multiple contracts. For our analysis, we collected 21,825 verified contracts by crawling `etherscan.io`, which we made publicly available [26]. We employed a web crawler to scan and retrieve the verified contracts’ source code with the retained layout and lexical features, which are critical for the source code authorship attribution. For each of the retrieved contracts, we also extracted the corresponding bytecode and disassembled it in order to obtain its opcodes. Our dataset contained a large number of authors with less than four contracts. Since having such a limited number of contracts limits the possibility of validating attribution results, we filtered out authors that had fewer than four contracts.

Number of authors	1071
Number of contracts	8915
Average contracts per author/ std/ median	8.32/ 12.19/ 5
Average source code byte size/ std/ median	114767.76/ 196095.32/ 61352
Average source code LOC/ std/ median	3330.93/ 5694.51/ 1821
Average bytecode byte size/ std/ median	185421.86/ 287453.45/ 119975
Minimum contracts per author	4
Maximum contracts of any author	256

Table 7.2: contracts’ dataset statistics

Table 7.2 shows the statistics of our resulting dataset, which contains 8915 contracts from 1071 authors with an average of 8.32 contracts per author. The contracts’ source code is on average 3330.93 Lines Of Code (LOC). Size-wise, the source code is on average 114767.76 bytes long and the bytecode is on average 185421.86 bytes long. All authors contain at least 4 contracts and the maximum number of contracts of any author is 256. For the relevant columns we also provide the standard deviation (std) and median values, which can help understand the amount of variation present in our data. For example, a high std value indicates that the samples significantly deviate from the average value and the median value represents a better estimate.

7.3.2 Evaluation results

The proposed attribution approach was implemented using the Python programming language with the scikit-learn module [107]. In the evaluation of our approach we use the accuracy, precision, and recall metrics, which are computed per author. Specifically, the *accuracy* shows the percentage of correctly classified code samples, for a given author (both correctly attributed to this author and not attributed to other authors) among all samples. The *precision* measures the percentage of correctly attributed code samples among all samples that were classified as a given author. The *recall (sensitivity to noise)* of a specific author is the probability of correct attribution, i.e., a percentage of correct code attributions to the number of times the given author was predicted. The overall macro-precision/macro-recall/accuracy are computed as an average of the precision/recall/accuracy respectively, over all

Classifier	Feature selection		
	RF	DT	MI
RF	✓	✓	✓
SVM			✓

Table 7.3: Inspected classifiers and feature selectors

authors. To ensure the reliability of these metrics, in this and the following sections we use a 4-fold stratified cross validation strategy where in each split (out of the 4 splits) the data are shuffled, 75% of the data are used for training, and 25% for testing. In this evaluation, we focused on several objectives:

1. Validating the effectiveness of benchmark attribution features for Ethereum contracts’ attribution.
2. Understanding the performance of our approach on a set of real-world contracts.
3. Exploring the ability of our approach to accurately attribute malicious contracts gathered in the wild.

7.3.3 Feature selection

In our study, we employ two feature sets that are widely used in code attribution studies: n-gram features, and a feature set derived by Caliskan et al. [43]. Both sets generate large and sparse feature vectors mostly due to their heavy use of unigram term frequencies. In many cases, such feature vectors lead to over-fitting. To avoid biased classification results, we examine three feature selection methods: Random Forest (RF) importance-based, Decision Tree (DT) importance-based, and Mutual Information gain (MI).

We classify the source code, bytecode, and Clasikan feature sets, which are extracted from the *selected4contracts* dataset (Table 7.2) using the configuration shown in Table 7.3. Figure 7.3 shows the classification accuracy for the different classifiers with their corresponding feature selectors. The RF classifier uses cumulative top

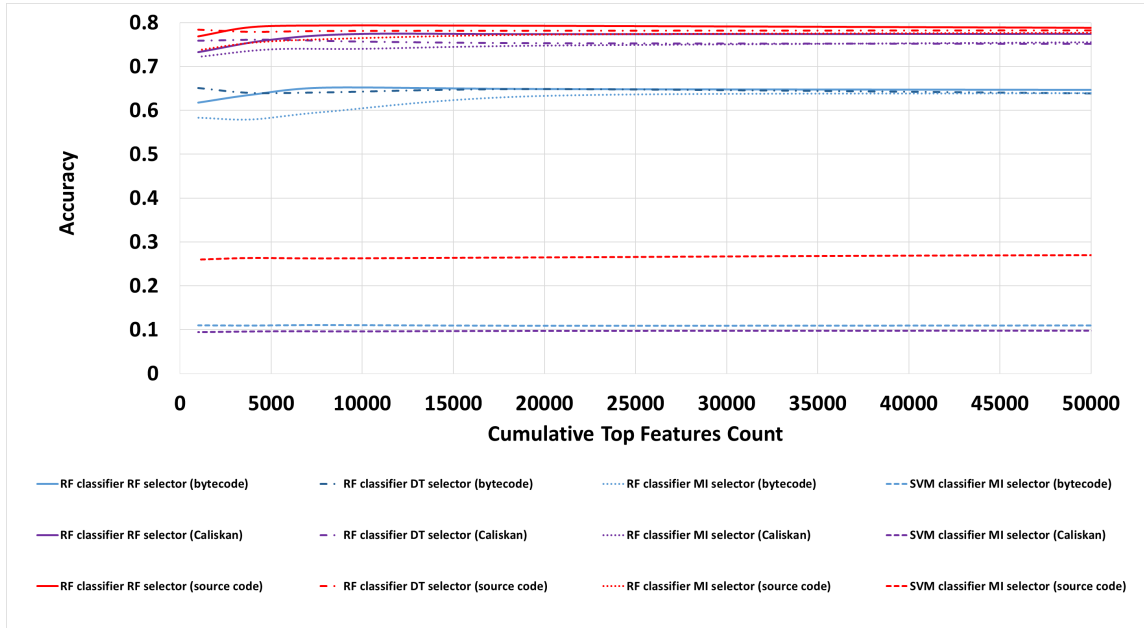


Figure 7.3: Preliminary classification accuracy using RF and SVM classifiers using cumulative top features that are extracted by RF, DT, MI feature selectors

ranked features, which are increasingly extracted using each of the RF, DT, and MI feature selectors. The SVM classifier uses cumulative top ranked features, which are increasingly extracted using the MI feature selector. It is apparent that the performance of the SVM classifier is significantly lower compared to the RF classifier on all feature sets. Research in authorship attribution that uses an SVM classifier often handles large samples of data. The samples in the *selected4contracts* dataset are composed of contracts’ code (source code or bytecode), that is characterized by a relatively small size, which may explain the effectiveness of the SVM classifier. Hence, we focus our analysis solely on the RF classifier.

Table 7.4 shows the RF classification accuracy using the RF feature selector (under the grouped column “RF feature selection”). When using the RF classifier, the DT importance-based feature selector provides the highest accuracy for the first 2000 features. When using more than 2000 features, the RF importance-based feature selector provides the highest classification accuracy, which is more than 50% higher compared to using the MI feature selector. We therefore continue to look at only DT

Program type	Feature set	RF feature selection		RF minimal top ranked feature selection		DT minimal top ranked feature selection	
		Total features	Accuracy	Top ranked features/ (ratio)	Accuracy	Top ranked features/ (ratio)	Accuracy
Source code	Unigram	231553	78.04%	8000 (3%)	79.04%	1500 (0.65%)	79.30%
Source code	Caliskan	441963	76.52%	10000 (2%)	77.49%	2000 (0.45%)	76.11%
Byte code	Unigram (of opcode)	63363	64.05%	8000 (13%)	65.00%	1500 (2.4%)	65.11%

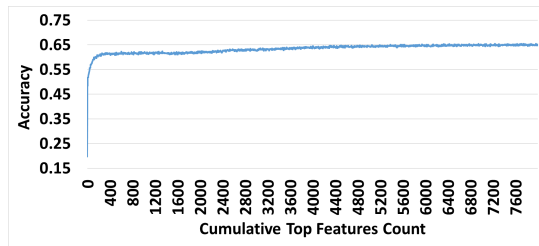
Table 7.4: RF classification accuracy per feature selector

and RF importance-based feature selectors for the RF classifier and their contribution to further feature count reduction in our analysis.

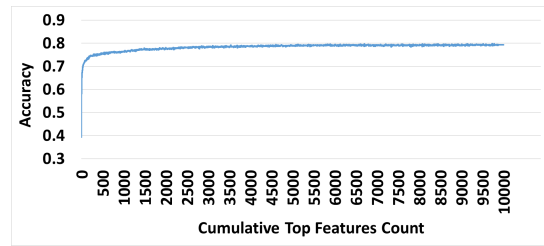
Top K ranked feature sets The classification accuracy of the RF classifier with the RF feature selector uses all available features, which can lead to overfitting. To reduce the overfitting we apply feature dimensionality reduction using the following method. We consider the classification accuracy of the RF classifier with the RF feature selector over all available features as the baseline accuracy which we would like to approximate with fewer features. For each feature selector, we incrementally select the top K ranked features and calculate the classification accuracy.

Figure 7.4 and Figure 7.5 show the change in accuracy per cumulative top feature count, which are incrementally extracted from the RF and DT feature selectors respectively, using the RF classifier. It is clear from the analysis that the accuracy quickly plateaus for all feature sets. Further, the highest accuracy is obtained with substantially fewer features when using the DT feature selector.

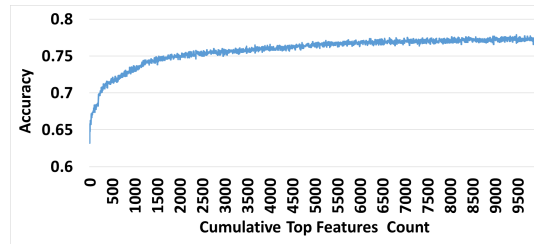
The best trade-off between the number of features (hence the size of the corresponding feature vectors) and accuracy is achieved at an earlier point of accuracy stabilization. When using the RF feature selector this stabilization point is reached at the 8000 top ranked features for opcode (bytecode) unigrams (13% of total features), 8000 top ranked features for source code unigrams (3% of total features),



(a) Bytecode unigrams

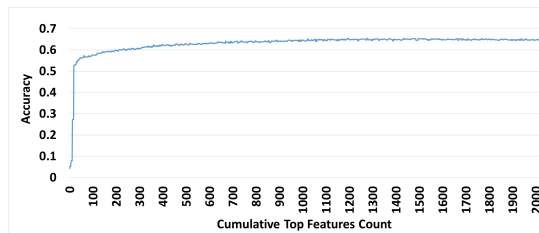


(b) Source code unigrams

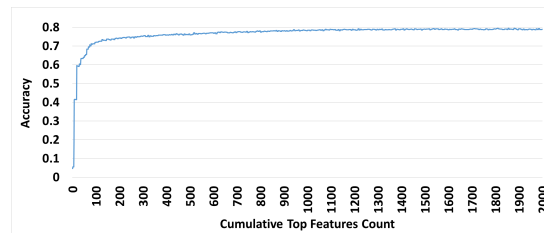


(c) Source code Caliskan

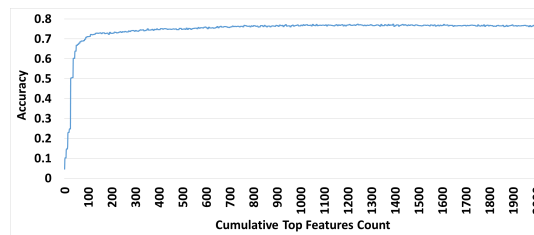
Figure 7.4: Finding cumulative classification accuracy of minimal top ranked features using a RF feature selector and classifier



(a) Bytecode unigrams



(b) Source code unigrams



(c) Source code Caliskan

Figure 7.5: Finding cumulative classification accuracy of minimal top ranked features using a DT feature selector with a RF classifier

and 10000 top features for the Caliskan feature set (2% of total features). When using the DT feature selector, the best trade-off occurs at 1500 top ranked features for opcode (bytecode) unigrams (2.4% of total features), 1500 top ranked features for source code unigrams (0.65% of total features), and 2000 top features for the Caliskan feature set (0.45% of total features). Table 7.4 summarizes the classification results, minimal top ranked features, and their ratio from the total number of features, per feature set and feature selector under the two last grouped columns. We can see that, although the accuracy does not change, the number of features retained for attribution analysis is significantly smaller.

Training and classification complexity The feature extraction using the RF/DT feature selectors, according to the specified ratios, and the RF classifier training are designed to be done offline. The cost to construct a DT feature selector, under the assumption that the sub trees are approximately balanced, is $\mathcal{O}(CF \log C)$ [27], where F is the number of features and C in the number of contracts. In a RF, multiple DT are trained, each with a subset of features. Hence, the cost is $\mathcal{O}(TCF \log C)$, where F is the number of the random subset of features per DT, C is the number of contracts, and T is the number of DTs in the RF. The cost of training a RF classifier is the same as constructing a RF feature selector. Querying a RF classifier is designed to be done online, with a cost of $\mathcal{O}(T \log C)$.

7.3.4 Heuristic refinement results

Table 7.13 details the different feature sets, which are extracted by the eight refinement heuristics (as discussed in Section 7.2.3) on the *selected4contracts* dataset. One noticeable difference between each heuristic is the amount of retained data. In *heuristics 5-8*, similarities are considered only between authors, which results in finding less common code between contracts and retaining more data. Contrarily, in

heuristics 1-4 similarities are considered between, as well as within authors, which results in finding more common code between contracts and hence retaining less data. A possible explanation is that authors are more likely to reuse their own code, e.g., to write and deploy different versions of the same contract, or to reuse their own components.

On one extreme, the least amount of code is retained with *heuristic 1*. This is the most conservative approach which removes a contract if it contains *any* similarities, where the similarities are searched between, as well as within authors. Note that for our analysis we choose to retain authors that have at least four contracts after applying any heuristic. On the other extreme, *heuristic 8* provides the highest data retention. In this approach, the similarities are looked for only between authors. Only contracts that are entirely composed of similar code are removed. The remaining contracts contain only dissimilar code splits. Each of the remaining splits is being treated as a “full contract,” which results in retaining more authors with at least four contracts. This heuristic can be used in cases where obtaining sufficient data are not feasible. Yet, even with this liberal approach, the resulting set contains only 658 authors, which are 61% of the authors in the original dataset.

The last two columns of Table 7.13 summarize the ratios of the top ranked features per feature set and feature selector (as discussed in section 7.3.3).

7.3.5 Attribution results

We perform the following steps per heuristic (as described in Section 7.2.3): refine the *selected4contracts* dataset (Table 7.2) as dictated by the heuristic, extract the feature sets using the corresponding feature selector and top ranked features’ ratios (Table 7.13). We next use the RF classifier on each of the refined feature sets. Table 7.14 shows the attribution accuracy, macro-recall, and macro-precision results per heuristic. The column “Minimal top ranked feature selection” contains the classifi-

cation accuracy that was achieved with the minimal top ranked features, per feature selector and feature set, as summarized in Table 7.4 under the last two grouped columns. We included the macro-recall and macro-precision in this column to facilitate more detailed comparison with the different heuristics' classification metrics. The consistent relatively low macro-recall results in this column may be related to the contracts' components similarities, as no data refinement was used, which can also explain the relative lower macro-precision results.

In most heuristics, the removal of similar code from the contracts results in a significantly higher macro-recall and macro-precision as compared to before applying the heuristics. As the changes in the macro-recall and macro-precision correlate with the changes in accuracy, we use the accuracy metric when comparing different heuristic results for brevity. The highest accuracy rates were obtained on *heuristic 5* with 93.25%/91.11% (RF/DT feature selector respectively) using the source code unigram feature set, and 91.68%/89.65% using the Caliskan feature set. The heuristics that produce partial contracts provide significantly more data. However, the type of analysis that can be performed is limited since no corresponding partial bytecode, or ASTs (used by the Caliskan feature set) can be extracted. These cases are labelled with a dash “-”.

Heuristic 5 shows the highest accuracy in the overall categories (excluding source code splits). It does this at the cost of removing all contracts that have any similarities, which results in the removal of many authors. *Heuristic 7* has the second-best performance. It removes only the similar components of the source code and merges the remaining components. Although it provides a slightly lower accuracy than *Heuristic 5*, it retains 6 times more authors.

Comparing heuristics, we see that caution should be exercised as to which similarities to remove. The single refinement property that distinguishes *heuristics 1-4* and *heuristics 5-8* is where to search for similarities. The former searches for similar-

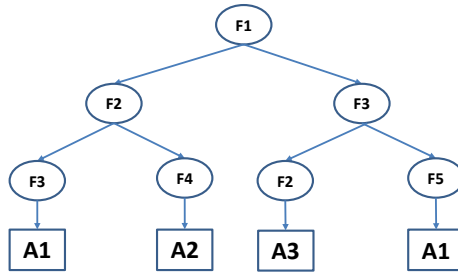


Figure 7.6: An example DT that was produced by the feature selector

ties between, as well as within authors while the latter searches for similarities only between authors. The preservation of similarities within the authors results in features that more accurately capture each author’s preference of using specific code constructs over others. Further, refining less data results in additional and larger samples, which also contributes to a higher classification accuracy.

While the approach that is based on the n-gram features provides the higher accuracy, it is also susceptible to manipulations of coding constructs’ names. In contrast, the approach that is based on the Caliskan features supports more robust features such as ASTs which can be more resilient in such cases at the cost of a slightly lower classification accuracy.

7.4 Exploring distinctly contributing features

The classification results summarized in Table 7.14 show that the accuracy, macro-recall, and macro-precision are similar when using either RF or DT feature selectors. The DT feature selector extracts significantly less features than are extracted by the RF feature selector (by an average factor of 4.8), while achieving similar accuracy. This implies that the RF feature selector selects unnecessary features. In the following experiments, we employ the RF classifier with the DT feature selector on the source code unigram and Caliskan feature sets.

Fig. 7.6 provides an example of a DT that was produced by the DT feature selector.

The nodes are of two types: *feature nodes* - contained in the inner nodes and prefixed with an F, and classified *authors nodes* - contained in the leaf nodes and prefixed with an A. Each feature node contains the feature to split the samples on and the impurity of the split in the form of a gini index. Each author node contains the classified author's address. Each feature contributes to the classification of at least one author. A feature can be used to classify multiple authors using multiple paths. For example feature F2 can classify authors: A1 (path F1-F2-F3), A2 (path F1-F2-F4), and A3 (path F1-F3-F2). From the example we see that author A1 can be classified using different paths. Feature F1 participates in classifying all authors, while feature F4 participates in classifying only author A1. With the help of this observation we next describe how to extract *distinctly contributing* features. To do so, we consider only features with nodes that exhibit a change in impurity, compared to their parent node's impurity, that is bigger than 1% (chosen empirically). We sort these features' list according to how many authors they participate in classifying, where features that participate in classifying less authors are higher. The resulting features list contains the distinctly contributing features at the top. The implementation of this approach is shown in Algorithm 3 and its dependent algorithms: Algorithm 1 and Algorithm 2.

Algorithm 1: `get_features_paths_per_author`

```

input : node - A DT root node, author_features_paths - feature paths per author (populated
          recursively), feature_data_list - features data list in current path (populated recursively)
1 if node  $\neq$  leaf node then
2   | feature_data_list.append((node.feature_name, node.impurity));
3   | get_features_paths_per_author(node.left, feature_data_list);
4   | get_features_paths_per_author(node.right, feature_data_list);
5 else
6   | author_features_paths[node.author_name].append(feature_data_list);

```

Algorithm 1 is the first dependent algorithm. This recursive algorithm traverses each path of a DT and extracts, per author, all the paths of features that lead to classify that author (a single author classification can be reached by several paths). The inputs to this algorithm are: *node* - the root of the DT; *authors_features_paths* - a

dictionary that will be populated with all features' paths per author and will serve as the output of the algorithm; *feature_data_list* - a helper variable that will be recursively populated with the current features' path. The algorithm traverses the DT using depth first search. As it traverses each node, a tuple containing the node's feature name and impurity is added to *feature_data_list*. When a leaf node (author address) is reached, the features' path, which is contained in *feature_data_list* is added per that author's address.

Algorithm 2: *get_authors_count_per_feature*

```

input : author_features_paths - features paths per author
output: feature_authors_count
1 feature_authors_count  $\leftarrow$  dictionary();
2 for (author_name, features_paths) in author_features_paths do
3   for features_path in features_paths do
4     for feature in features_path do
5       feature_authors_count[feature.feature_name].add_to_set(author_name);
6 for (feature_name, authors_set) in feature_authors_count do
7   feature_authors_count[feature_name]  $\leftarrow$  len(authors_set);
8 return feature_authors_count

```

Algorithm 2 is the second dependent algorithm. It extracts, per feature, the number of distinct authors that this feature helps to classify. The only input to this algorithm is the *author_features_paths* that was populated by Algorithm 1. We start by iterating over all the features in all the authors' paths and populate *feature_authors_count* with a set of distinct authors (and without duplicates) per the feature which appears in any path that leads to their classification. The last for-loop of the algorithm replaces the distinct authors' set, per feature, with the set's size.

Algorithm 3 is the main algorithm. It produces a list of all distinctly contributing features that are used in the classification of any author as specified by the DT. The only input to this algorithm is *DT* - the DT as was generated by the DT feature selector. It uses Algorithm 1 to populate *author_features_paths* and Algorithm 2 to populate *feature_authors_count*. It continues to iterate over all the features in all the authors' features' paths. In each path the current feature impurity is compared to the previous feature impurity. We sum the impurities where the change is greater

Algorithm 3: *get_ranked_distinctly_contributing_features*

```
input : DT - The DT as was generated by the DT feature selector
output: distinctly_contributing_features_list
1 author_features_paths  $\leftarrow$  dictionary();
2 get_features_paths_per_author(DT.root, author_features_paths, list());
3 feature_authors_count  $\leftarrow$  get_authors_count_per_feature(author_features_paths);
4 feature_impurity_change_list  $\leftarrow$  dictionary();
5 for (author_name, features_paths) in author_features_paths do
6   for features_path in features_paths do
7     prev_impurity  $\leftarrow$  NULL;
8     for feature in features_path do
9       if prev_impurity is NULL then
10        | feature_impurity_change_list[feature.feature_name] = list( $\infty$ );
11       else
12        | impurity_change  $\leftarrow$  abs((feature.impurity - prev_impurity)/prev_impurity);
13        | if impurity_change > 0.01 then
14        | | feature_impurity_change_list[feature.feature_name].add(impurity_change);
15        | prev_impurity  $\leftarrow$  feature.impurity;
16 distinctly_contributing_features_list  $\leftarrow$  list();
17 for (feature_name, impurity_change_list) in feature_impurity_change_list do
18   authors_count  $\leftarrow$  feature_authors_count[feature_name];
19   feature_impurity_change_list  $\leftarrow$  impurity_change_list[feature_name];
20   ranked_feature_impurity_change  $\leftarrow$  sum(feature_impurity_change_list)/authors_count;
21   distinctly_contributing_features_list.add((feature_name, ranked_feature_impurity_change));
22 // sort distinctly_contributing_features_list descending by ranked_feature_impurity_change
23 return distinctly_contributing_features_list
```

than 1%, per the feature’s name (the first feature in each path, which does not have a predecessor, is assigned a purity change of ∞). Following this for-loop we calculate, per feature, the ratio of the feature’s sum of impurity changes to the number of distinct authors that this feature can be used to classify. The reason for using this ratio is that the sum of impurity changes captures the degree to which this feature distinctly contributes to the classification of the authors that are part of its path; the larger the impurity changes (*feature_impurity_change_list*) sum is, the more distinct the contribution is. Conversely, the more distinct authors the feature can classify (*authors_count*), the less distinct is its contribution. Note that, per feature, the number of impurity changes that did not pass the 1% threshold may be lower than the overall number of authors that this feature can help to contribute to, which is why the overall authors count was used.

The overall distinctly contributing features can be computed for all authors in a feature set, as we explore next, and can be computed for a specific author, which is

Program type	Feature set	Overall num. of features	Overall num. of distinctly contributing features	Overall num. of distinctly contributing features/ Overall num. of features
Source code	Unigram	319	42	13%
Source code	Caliskan	313	49	16%
Bytecode	Unigram (of opcode)	461	99	21%

Table 7.5: Statistics of distinctly contributing features for source code unigram, bytecode unigram, and Caliskan feature sets encompassing all authors (*selected4contracts* dataset was used with a *Heuristic 5* refinement)

Feature	Ranking value
information:	∞
_required	0.8369565217391308
adding	0.7784671177308304
host.	0.71670031753775
Migrations(new_address);	0.6690783182192911
payable	0.6686851148489782
for	0.649206562516811
_amount;	0.6248371689101172
Check	0.624346349239456
with	0.5859452155731127
before;	0.5837308057551378
developerWallet);	0.5535714285714286
if(_what	0.5393504682207612
0.4.21;	0.5221321196027492
why	0.49697338734990015
(uint8	0.4937226277372263
don't	0.49281487743026203
account;	0.4898230848505668
by	0.4879104569909097
bytes32	0.40740740740740744

Table 7.6: Top 20 of 42 most distinctly contributing source code unigrams encompassing all authors (a DT feature selector was used on the *selected4contracts* dataset, which was refined with a *Heuristic 5* refinement)

explored in Section 7.5.1. Table 7.5 shows a summary of the number of the distinctly contributing features encompassing all authors and how they compare to the total number of features of all authors for the source code unigram, bytecode unigram, and Caliskan feature sets. The ratio of “Overall num. of distinctly contributing features” to “Overall num. of features” is dependent on the impurity change threshold chosen. We continue to further examine the features for the source code unigram and Caliskan feature sets. The feature sets were extracted with a DT feature selector following the use of *Heuristic 5* to refine the dataset, which provides the highest attribution

Feature	Ranking value
1-information:	∞
5-ExpressionStatementContext	0.8369565217391308
10-consonants	0.7992626728110599
9-ledger	0.7783705308710431
1-/***/function	0.7780791339119371
4-PragmaDirectiveContext	0.7242677375005225
10-private	0.7092657173302335
9-realDevReward	0.6959175084175084
6-MappingContext	0.6489308348151188
9-send	0.6306329685283942
3-TypeNameContextmypackage.SolidityParser\$ElementaryTypeNameContext	0.5991649269311065
1-other._len);	0.5832459151885053
10-bool	0.5683561062123748
1-sorted	0.5215305162826198
3-StatementContextmypackage.SolidityParser\$ReturnStatementContext	0.5061728395061729
1-enough	0.49626732892080294
10-_to	0.4924392771211296
9-account	0.4891791044776119
7-returns	0.4646017699115044
3-FunctionDefinitionContextmypackage.SolidityParser\$ReturnParametersContext	0.4402370760619032

Table 7.7: Top 20 of 49 most distinctly contributing source code Caliskan features encompassing all authors (a DT feature selector was used on the *selected4contracts* dataset, which was refined with a *Heuristic 5* refinement)

Prefix	Feature type
1-	WordUnigramTF
2-	numKeyword
3-	ASTNodeBigramsTF
4-	ASTNodeTypesTF
5-	ASTNodeTypesTFIDF
6-	ASTNodeTypes ExcludingLeavesAvgDepths
7-	solidityKeyWords
8-	CodeInASTLeavesTF
9-	CodeInASTLeavesTFIDF
10-	CodeInASTLeavesAvgDep

Table 7.8: Caliskan features legend

accuracy and supports all the required feature sets. Table 7.6 shows the top 20 most distinctly contributing features for the source code feature set. Table 7.7 (and its corresponding Table 7.8) shows the top 20 most distinctly contributing features for the Caliskan feature set.

	Num. of contracts	Avg source code LOC / (std)	Avg source code byte size / (std)
Ponzi	148	125.03 / (120.68)	3894.92 / (4642.51)

Table 7.9: Ponzi scheme dataset statistics

Program type	Feature set	Overall num. of features	Overall num. of distinctly contributing features	Overall num. of distinctly contributing features / Overall num. of features	Ponzi author num. of distinctly contributing features
Source code	Unigram	2291	392	17%	39

Table 7.10: Statistics of distinctly contributing source code unigrams encompassing all authors and the Ponzi author specifically (a DT feature selector was used on the combination of the *selected4contracts* and *Ponzi* datasets refined with *Heuristic 7*)

7.5 Real-world scams

7.5.1 Attributing Ponzi scheme smart contracts and examining their distinctly contributing features

Victims who invest in the contract scams may assume unjustifiably that the fact that a contract is being used implies that a fair execution is enforced due to the lack of central authority management, exchange of funds are publicly available, which increases trustworthiness, or that the immutability of the data ensures higher security. In this section we explore a Ponzi scheme dataset containing 148 Ethereum contracts provided by Bartoletti et al. [35]. Table 7.9 shows the statistics of the Ponzi scheme dataset. We group the Ponzi scheme contracts under a single author that we call “Ponzi,” add this author to the *selected4contracts* dataset, and refine it using *heuristic 7* (this second best performing heuristic contains significantly more authors than the best performing heuristic). We continue to perform the feature selection using a DT feature selector and classify the feature sets using a RF classifier with an accuracy of 86.12%, recall of 83.67%, and precision of 81.9%. In addition, we extract the distinctly contributing features from the source code unigram feature set using the approach presented in section 7.4. Table 7.10 shows the summary of

Feature	Ranking value
TokenDone.io	∞
SimpleDice()	1.7974461185055057
bribedCitizen.send(address(this).balance);	1.668637138489296
_newAddress)	0.6428571428571429
payable	0.3917320237963122
BalancedPonzi()	0.3607672195900195
i=0;	0.3233214474257536
if(0.2860334132002368
NiceGuyPonzi()	0.26122448979591834
multiplier;	0.26068836045056315
last	0.1951354339414041
or	0.15743440233236147
+=	0.13685636856368563
publicKeyPart));	0.1363636363636364
lastDepositor.send(msg.value);	0.12500000000000003
9/10	0.12500000000000003
_contractAddress;	0.1
Coinflip	0.08799999999999997
false;	0.06970025881471757
time	0.05357142857142862

Table 7.11: Top 20 of 39 most distinctly contributing source code unigrams of the Ponzi author (a DT feature selector was used on the combination of the *selected4contracts* and *Ponzi* datasets refined with *Heuristic 7*)

the distinctly contributing features encompassing all authors under column “Overall num. of distinctly contributing features,” which comprise 17% of the “Overall num. of features.” The last column presents the number of the distinctly contributing features, which are related to the Ponzi author. Table 7.11 shows the top 20 of 39 most distinctly contributing features of the source code unigram feature set, which are used in a Ponzi scheme attribution, where the most distinctly contributing features are at the top. We can see features that contain the words “Dice,” “bribed,” and “Ponzi,” for example, which are intuitively related to scamming contracts in general and to Ponzi scheme contracts in particular.

7.5.2 Attributing real-world scammers

The results show the effectiveness of our approach for attributing unknown contracts to their corresponding authors. To examine our approach in the underground scammers’ community, we further explore the attribution of real-world Ethereum scams. Etherscamdb [22] is an open source database that keeps track of the current

Ethereum scams. The scam information in the site is diverse and contains scam categories like “Fake ICO”, “Phishing”, “Scamming”, and “Scam”. We extract contracts based on the account address and the contract’s address. For a given malicious contract’s address, we extract all contracts deployed by the contract’s deployer. Sym-

Algorithm 4: Retrieve scammers data from EtherscamDB

input : $ScmDB_{addresses} = (a_1, a_2, \dots)$ - Etherscamdb’s malicious addresses, $Scm_{authors} = (au_1, au_2, \dots)$
- contract malicious authors. Each author contains $SC = (sc_1, sc_2, \dots)$ were sc_i is a contract’s address

output: $Scm_{authors}$

```

1  $Scm_{authors} \leftarrow \{\}$ ;
2  $A \leftarrow \{\}$ ;
3 for  $a_i$  in  $ScmDB_{addresses}$  do
4   if  $a_i$  is account address then
5      $A.Add(a_i)$ ;
6   else if  $a_i$  is contract’s address then
7      $T \leftarrow etherscan.get\_all\_transactions(a_i)$ ;
8      $a_{addr} \leftarrow get\_contract\_creator\_account\_address(T)$ ;
9      $A.Add(a_{addr})$ ;
10 for  $a_i$  in  $A$  do
11    $T \leftarrow etherscan.get\_all\_transactions(a_i)$ ;
12    $a_{contracts} \leftarrow get\_contract\_creation\_addresses(T)$ ;
13    $Scm_{authors}.Add(\{a_i, a_{contracts}\})$ ;
14 return  $Scm_{authors}$ 

```

metrically, for each malicious account address, we extract all its deployed contracts (if any) and consider them to be malicious (see Algorithm 4). To fetch the account address of a malicious contract’s deployer or the contract addresses deployed by a malicious account address, we use `etherscan.io`. To implement the algorithm, for each address that was extracted from Etherscamdb, we use `etherscan.io`’s API: `api.etherscan.io/api?module=account&action=txlist&address=< address >` to find the transactions that are related to a requested address. We provide a malicious address and receive a transaction list. For each transaction we examine the “results” field. An empty “to” field indicates that the transaction is used to deploy a contract whose address is provided in the “contractAddress” field. When the “contractAddress” is the same as the provided malicious address, this indicates that the provided address is a contract address. If they are different, this indicates that the provided malicious address is an account address and the “contractAddress” value is the de-

	Num. of authors	Num. of contracts	Max contracts per author	Avg contracts per author	Avg source code byte size	Avg source code LOC
Source code/ Bytecode	22	68	20	3.09	7379.98	212.91

Table 7.12: Scammers dataset statistics

ployed contract’s address. If the “results” field is empty, an account or contract was not found at the specified address and is ignored.

The algorithm’s result set contains all authors related to malicious contracts with their deployed contracts’ addresses. For each of the contract’s addresses, we extract its opcode and source code (if available in `etherscan.io`’s verified contracts). The resulting scammers dataset statistics are shown in Table 7.12. We cross referenced the scammers dataset and our *selected4contracts*, presumably, benign dataset. One account address (`0x0042bd345e43bd151fa563c2bc8fa22bda507104`) was contained in both datasets. It contained 8 verified contracts in our collected scammers dataset, of which 5 contracts were found in our *selected4contracts* benign dataset. The 5 shared contracts as well as the 3 unseen contracts were attributed correctly in the source code and bytecode dataset.

7.6 Comparison to related work

We compare our proposed approach to the surveyed approaches in the related work. Most surveyed approaches related to blockchain address affiliation (Section 3.2.4) heavily rely on unverified assumptions. For example, addresses are determined to be associated with the same user if they share a transaction [44], transfer remaining funds to other addresses [127] (i.e., change addresses in bitcoin), or used as inputs in the same transaction [99,113]. However, these approaches cannot be validated due to the blockchain accounts’ pseudo-anonymity that prevents researchers from verifying which addresses belong to the same user. As a result, no dataset is available to validate these approaches. In our proposed approach, we use a conservative approach

and assume that each user is identified by a single account address, which can be used to deploy multiple contracts. Using this approach, we prepared a real-world dataset that contains all addresses that were used to deploy contracts and their related Solidity code and bytecode. Further, our evaluation on this dataset validated that our approach can accurately classify contract code to their deployers' addresses using stylometry techniques, which in turn can affiliate multiple deployers' addresses that were used to deploy contracts' code with similar characteristics.

The approach taken by Norvil et al. employs unsupervised clustering based on the *ssdeep* hash similarity [86] of Ethereum bytecode. While hash similarity employs contract code string comparisons, and hence can be sensitive to code structure changes, our proposed approach considers similarities of coding style characteristics. Attribution based on stylistic characteristics can provide high classification accuracy for contract codes that seem very different, as long as the stylistic features are similar. In addition, as the results of our work suggest, code reuse is prevalent in contracts' code, which can contribute to hash similarity bias. We further make use of heuristics that refine code similarities to reduce the attribution bias.

The approaches to de-anonymize blockchain addresses (Section 3.2.5) make use of out-of-network information and are complementary to our work. Our approach uses both out-of-network Solidity source codes and the assumption that each code author possesses a distinctive writing style that is reflected in the contract code they write. These are used to cluster affiliated account addresses that were used to deploy contracts based on their contracts' code, using stylometry techniques. Further, once account addresses are clustered, de-anonymizing only a few addresses in the cluster can de-anonymize the remaining clustered account addresses.

In the related work regarding authorship attribution (Section 3.2.6), we discuss the state-of-the-art method proposed by Caliskan et al., which adopts ASTs and unigrams term frequency for attribution of code. In this work, we use a similar feature

set, modified to support the Solidity source code, which we call “Caliskan features.” In the study by Watson et al. [135], the authors aimed to attribute C++ programmers who contributed to a Github repository. They used three feature extraction methods: character-level, token-level, and AST-level. The features were extracted from commit statements at the function level. For the attribution model the random-forest classifier was used. While this approach’s domain is different than attribution of Solidity source codes, the function granularity that was taken results in relatively small samples closely resemble smart contracts code samples. The authors achieved a source code attribution F1 score of 0.75 on a dataset comprised of 346 authors, each with an average 234 functions. In comparison, our approach achieved a unigram source code attribution F1 score of 0.73 on a dataset comprised of 1071 authors each with an average 9 contracts *before heuristics refinement*. While the attribution results are nearly identical the dataset vary widely with $3\times$ more authors and $26\times$ less samples per author in our approach compared to the authors’ approach. However, since the two approaches are similar in that the authors extract features that contain unigrams and use a random-forest classifier, similar to our approach, the primary distinction is our approaches is the use of refinement heuristics. After *heuristic 5* refinement, we achieved a unigram source code attribution F1 score of 0.90 on a dataset comprised of 106 authors each with an average 9 contracts; and after *heuristic 7* refinement, we achieved a unigram source code attribution F1 score of 0.85 on a dataset comprised of 599 authors each with an average 8 contracts. It can be interesting to verify if such a heuristics approach would also contribute to a higher attribution rate in the authors approach on C++ source codes. The work by Bartolletti et al. [35], which analyzed Ethereum contracts involved in a Ponzi scheme used redistribution of payouts to locate the Ponzi scheme-related contracts. Our approach is complementary in nature to this approach and can help determine the scammers behind the scheme. To the best of our knowledge, no studies were conducted on

clustering or de-anonymizing Ethereum addresses using authorship attribution on Ethereum contracts' code.

H	Program type	Feature set	Retained data %	Num. of authors	Avg contracts per author/(std)	Avg contract byte size/(std)	Avg contract LOC/(std)	Top/total features (RF feature selector)	Top/total features (DT feature selector)
1	Bytecode	Unigram	1.60%	34	6 / (2.48)	94371 / (80672.32)	-	1500 / 11590	280 / 11590
	Source code	Unigram	1.25%	34	6 / (2.48)	45162.41 / (53276.11)	1208 / (1345.128)	650 / 21550	145 / 21550
	Source code	Caliskan	1.25%	34	6 / (2.48)	45162.41 / (53276.11)	1208 / (1345.128)	700 / 34987	160 / 34987
2	Source code splits	Unigram	1%	76	6.88 / (3.433)	22917.43 / (26892.87)	663.67 / (761.45)	1000 / 29623	195 / 29623
	Bytecode	Unigram	21.50%	283	6.24 / (3.723)	149305.82 / (127928.95)	-	6000 / 42768	1030 / 42768
3	Source code	Unigram	9.50%	283	6.24 / (3.723)	41467.36 / (39868.53)	1186 / (1070.32)	5000 / 135563	885 / 135563
	Source code splits	Unigram	13%	475	9.65 / (7.893)	33281.7 / (34426.26)	958.05 / (929.51)	6000 / 175427	1145 / 175427
4	Bytecode	Unigram	7.70%	106	8.39 / (9.173)	144171.77 / (195275.03)	-	3000 / 21422	515 / 21422
	Source code	Unigram	6.50%	106	8.39 / (9.173)	76077.88 / (142905.22)	2091.55 / (3900.64)	1000 / 49030	320 / 49030
	Source code	Caliskan	6.50%	106	8.39 / (9.173)	76077.88 / (142905.22)	2091.55 / (3900.64)	3000 / 91315	415 / 91315
6	Source code splits	Unigram	6.60%	136	15.48 / (23.673)	59369.63 / (127319.81)	1660.75 / (3498.86)	1600 / 52515	345 / 52515
	Bytecode	Unigram	54.60%	599	7.66 / (7.028)	180939.86 / (212620.36)	-	8000 / 57508	1385 / 57508
7	Source code	Unigram	32.23%	599	7.66 / (7.028)	66135.62 / (108244.7)	1894.51 / (3138.6)	6000 / 196478	1280 / 196478
	Source code splits	Unigram	33.40%	658	17.94 / (29.373)	61984.62 / (104176.03)	1787.22 / (3040.22)	7000 / 203899	1325 / 203899

Table 7.13: Feature set per heuristic refinement statistics

	Program type	Feature set	Minimal top ranked feature selection	Heuristic (accuracy% / macro-recall% / macro-precision%)							
				1	2	3	4	5	6	7	8
RF feature selection	Bytecode	Unigram	65% /	54.41% /	58.55% /	-	80.88% /	75.08% /	-	70.91% /	-
			56.04% /	52.08% /	55.70% /	-	74.16% /	70.91% /	75.08% /	-	70.91% /
	Source code	Unigram	52.53%	44.34%	50.38%	-	73.06%	67.42%	-	88.06% /	-
			79.04% /	81.86% /	71.34% /	-	93.25% /	85.76% /	88.06% /	-	85.76% /
	Source code	Caliskan	74.21% /	80.39% /	68.70% /	-	90.33% /	83.72%	-	85.76% /	-
DT feature selection	Bytecode	Unigram	71.09%	76.31%	65.19%	-	89%	-	-	-	-
			77.49% /	77.94% /	68.07% /	-	91.68% /	87.64% /	91.68% /	-	87.64% /
	Source code splits	Unigram	72% /	77.20% /	63.19% /	60.50% /	87.64% /	90.02% /	78.46% /	87.25% /	79.14% /
			68.68%	72.95%	60.14%	54.56% /	85.31%	78.26%	78.26%	79.14% /	79.6%
	Source code	Caliskan	65.11% /	53.43% /	57.13% /	-	79.75% /	74.08% /	-	74.08% /	-
56.56% /			52.75% /	54.34% /	-	73.57% /	69.53% /	69.53% /	-	69.53% /	
Source code splits	Unigram	53%	45.01%	48.39%	-	70.61%	65.72%	-	65.72%	-	
		79.3% /	77.94% /	68.91% /	-	91.11% /	87.1% /	87.1% /	-	87.1% /	
Source code	Caliskan	74.41% /	76.22% /	65.76% /	-	86.63% /	84.62% /	-	84.62% /	-	
		71.2%	71.54%	61.18%	-	84.11%	82.6%	82.6%	-	82.6%	
Source code splits	Unigram	76.11% /	77.45% /	-	-	89.65% /	-	-	89.65% /	-	
		70.67% /	77.08% /	64.62% /	57.11% /	85.42% /	83.06%	83.06%	86.6% /	78% /	
67.29%	73.27%	60.54% /	49.94%	55.75%	55.75%	76.54%	76.54%	78% /	78.83%		

Table 7.14: RF classification results per feature selector, feature set, and heuristic

Chapter 8

Authenticated Multi-Version

Index for Blockchain-based Range

Queries on Historical Data

The rapid adoption of blockchain systems with contract support unraveled additional challenges and requirements. These include, stringent regulatory requirements that require detailed audit of traceable historical data to enforce accountability; and emerging security issues, which require efficient analysis of historical data to manage future security issues through methods such as transaction ordering dependency analysis [132] or execution of tainted data [95].

Current approaches to managing and querying historical data in existing blockchains rely on downloading the historical data and managing them off-chain [108, 118]. For example, a full blockchain archive node stores all executed transactions' historical data, as they are executed, in a dedicated database off-chain. In an alternative approach the historical data can be retrieved ad hoc by replaying all transactions, which are not efficient. However, downloading and managing historical data off-chain has its own challenges: (1) the downloaded data are not part of the consensus protocol.

As a result, an additional mechanism is required to ensure their tamper-resistance and *fidelity* to the blockchain data source in order to adhere to the regulatory requirements, (2) to enable querying the high volume of historical data, they first need to be downloaded and further indexed (using an external index). This can be quite slow and hence, not suitable for real time queries, (3) the historical data can only be accessed by nodes that actively manage their history, as a result they cannot be synchronized and authenticated using a consensus protocol between participating nodes to ensure the same data are shared. Consequently, the historical data cannot be consumed by contracts to enable richer online (and real-time) capabilities.

Managing historical data efficiently on-chain, as part of the consensus protocol, can address the above issues. However, the underlying blockchain structure does not provide efficient support for managing and querying historical data. Recently, researchers have proposed indexing structures to support efficient search over blockchain data such as Merkle Patricia Trie (MPT) [137] that is used in the Ethereum blockchain to manage state data, and Merkle Bucket Tree (MBT) [51] that is used in the Hyperledger blockchain to manage state data, which were subsequently evaluated by a systematic study [141]. For example, the MPT is designed for key-value lookups (where the key is a storage address and the value can have a varying data and size), which means it is designed to quickly find the value corresponding to a specific key (storage address). It is not optimized for scanning through a range of keys. This is because an MPT is organized as a hierarchical tree structure, where each node represents a portion of the key (or storage address) being searched. Each level of the tree corresponds to a different prefix of the key, with the root node representing the entire key. To perform a range search on an MPT, one would need to traverse the entire tree, checking each node to see if it falls within the range being searched. This can be computationally expensive, particularly for large ranges or deep trees. When a new storage value is stored in an MPT, the previous value is overwritten.

This means that the new value replaces the old value, and the old value is no longer accessible from the MPT. Once the value is updated the hash of all parent nodes are updated along the path to the root in order to ensure the integrity of the tree. Further the MPT root hash represented the authentication digest of the entire MPT, which is used to authenticate that the state of the tree matches its block expectation. It is important to note that the keys and values are retained only in the MPT data structure, which is maintained externally to the blockchain. Only the MPT root hash, which encodes the MPT's entire state per the specified block is stored in the corresponding block on the blockchain. The MBT index behaves similarly. For these reasons, the proposed indexes only support point queries on the latest data and do not maintain historical storage values for versioning purposes. Hence, more advanced querying capabilities that can enable efficient analysis of large amounts of authenticated historical data, both in offline and online (e.g., through contracts) settings, are challenging with current blockchain data structures.

Goodrich et al. [76] proposed the idea of authenticated skip list, in which the authentication process relies on the root digest, a single hash value, that encapsulates the hash values of all the skip list nodes. Any change to the skip list invalidates the root digest and requires its recomputation, which requires re-traversing all of the skip list nodes recursively. Inspired by this approach, we present the Authenticated Multi-Version Skip List (AMVSL) index. AMVSL helps to address the aforementioned challenges by providing advanced querying capabilities over current and historical authenticated data as well as efficient versioned data management, which can naturally be applied to blockchain-based systems. It enables online data querying and modification (e.g., by contracts) as well as offline querying (e.g., by regulatory bodies). Additionally, AMVSL can be synchronized between nodes using a consensus protocol in the same way current authenticated data structures such as MPT and MBT are used. AMVSL is designed to support the following data management and

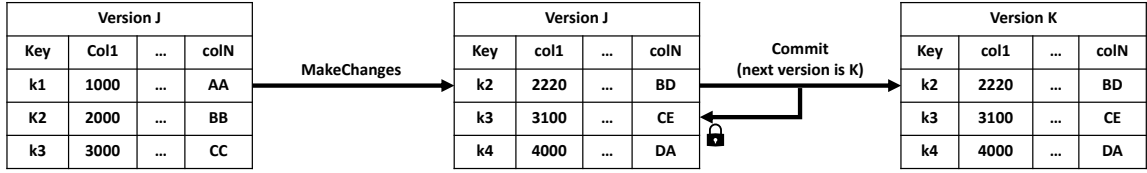


Figure 8.1: Versioned relational schema for a table T

querying use-cases:

- Insert, update, delete, and commit versioned data.
- Efficient range queries on keys and their historical values.

To support these use-cases we introduce three range queries (formally defined in Section 8.1): (i) Single Version on Ranged Keys (SVRK) query, (ii) Multi-Version on Ranged Keys (MVRK) query, and (iii) Multi-Version on All Keys (MVAK) query. With these query features, AMVSL can be used as a building block for developing a blockchain-based database system with a support for versioned relational schema. In a broader sense, AMVSL enables users to issue temporal range queries across multiple versions of a table. As an example, consider a general relational table T (version J , Figure 8.1). Several row insert, update, and delete operations will result in a modified table. If this current snapshot of the table T is required to be preserved (e.g., before/after an entire blockchain block is executed), a commit operation will generate a new version K , consequently preserving a previous state of the table as version J . The subsequent row insert, update, and delete operations will not affect the table’s previously committed versioned snapshots (e.g., version J); thus, creating immutable snapshots. These snapshots present historical execution data and can be efficiently queried with the use of multi-version range queries.

To evaluate the proposed AMVSL index, we implemented a prototype system that includes AMVSL, along with the blockchain-based range queries: SVRK, MVRK and MVAK. Example query scenarios that can be facilitated by these queries include: (i) querying the latest value of a key or a range of keys; (ii) querying historical values of

a key or a range of keys, e.g., analysis involving running averages of current states or analysis of the change in the current Ether amount compared to a historical amount; (iii) extracting all index keys' values in a version or a range of versions, e.g., querying over a period of time (when versions are derived from timestamps), or querying over a range of transactions/blocks (when versions are derived from transactions/blocks numbers).

For comparative analysis, we also implemented two existing state-of-the-art indexes, MPT and MBT. Since these indexes do not directly support versioning or range queries, we modified them to support these capabilities. To summarize, the main contributions of our work are as follows:

- We propose a novel authenticated multi-version index AMVSL to support efficient blockchain-based historical data management and querying.
- We propose and implement three blockchain-based range queries: SVRK, MVRK and MVAK.
- With extensive experimental evaluation, we demonstrate that AMVSL significantly outperforms MPT and MBT in the query execution benchmarks. In the best case, we achieve a speedup of $26\times$ over these approaches. AMVSL also shows superior or comparable performance in insert and update operations.

8.1 Problem definition

A verifiable index, which supports versioning as well as range queries should enable performing range queries on keys and versions. To test and evaluate these capabilities, we propose three multi-version and single-version range queries. This section outlines the key definitions that set the ground for the range queries.

Let $Keys$ be the set of keys of size $|Keys|$, $Vers$ be the set of versions of size $|Vers|$, and $Vals$ be the set of values. We define a multi-version index I as the set of

Query Name	Query description
SVRK	$query(version, keyStart, keyEnd)$
MVRK	$query(verStart, verEnd, keyStart, keyEnd)$
MVAK	$query(verStart, verEnd)$

Table 8.1: Multi-version and single-version range queries

comprising tuples in the form $(key, version, value)$, as defined in Eq. 8.1.

$$I \subseteq \{Keys \times Vers \times Vals\} \quad (8.1)$$

Let $OKeys$ be an ordered list comprising of the keys in $Keys$ in an ordered manner and $OVers$ be the list of the versions in $Vers$ in an ordered manner.

Let $ROKeys_{keyStart}^{keyEnd}$ be an ordered list of sequential keys in $OKeys$ in the range $[keyStart, keyEnd]$ as defined in Eq. 8.2.

$$ROKeys_{keyStart}^{keyEnd} \equiv (k \in OKeys | keyStart \leq k \leq keyEnd) \quad (8.2)$$

$ROVers_{verStart}^{verEnd}$ is defined similarly in Eq. 8.3

$$ROVers_{verStart}^{verEnd} \equiv (v \in OVers | verStart \leq v \leq verEnd) \quad (8.3)$$

We define the following multi-version and single-version range queries, which are summarized in Table 8.1.

Definition 8.1.1. *The query **Single Version on Ranged Keys** (SVRK) retrieves the values of the corresponding keys in the range $[keyStart, keyEnd]$ that are valid in the given single version as defined in Eq. 8.4. It is expressed as $query(version, keyStart, keyEnd)$.*

$$SVRK \equiv (value | (key, version, value) \in I \wedge key \in ROKeys_{keyStart}^{keyEnd}) \quad (8.4)$$

Definition 8.1.2. The query **Multi-Version on Ranged Keys (MVRK)** retrieves the values of the corresponding keys in the range $[keyStart, keyEnd]$ on the versions in the range $[verStart, verEnd]$ as defined in Eq. 8.5. It is expressed as $query(verStart, verEnd, keyStart, keyEnd)$.

$$MVRK \equiv (value|(key, version, value) \in I \wedge key \in ROKeys_{keyStart}^{keyEnd} \wedge version \in ROVer_{verStart}^{verEnd}) \quad (8.5)$$

Definition 8.1.3. The query **Multi-Version on All Keys (MVAK)** retrieves all values of the corresponding versions in the range $[verStart, verEnd]$ for all keys as defined in Eq. 8.6. It is expressed as $query(verStart, verEnd)$.

$$MVAK \equiv (value|(key, version, value) \in I \wedge version \in ROVer_{verStart}^{verEnd}) \quad (8.6)$$

8.2 Proposed approach

Existing blockchain verifiable index implementations enable point query on a given key in the latest (current) snapshot. Our focus in this work is advanced querying capabilities across versions. The index snapshot's *version* can be defined in any desired granularity, i.e., each row insert, update, and delete operation may define a version, similarly, a batch encapsulating a single operation of multiple rows may define a version; a transaction, which encapsulates multiple queries may define a version, etc. The version can be of any type that enforces a strictly increasing order, e.g., date, time (e.g., the timestamp that is assigned to each block on its creation), running sequential number (e.g., block number), and ordered string.

Our proposed approach enables the aforementioned capabilities using a verifiable data structure, called AMVSL, which consists of a *skip list-based index* for the table's keys, and a *multi-version list-based buckets* (partitions) for their corresponding

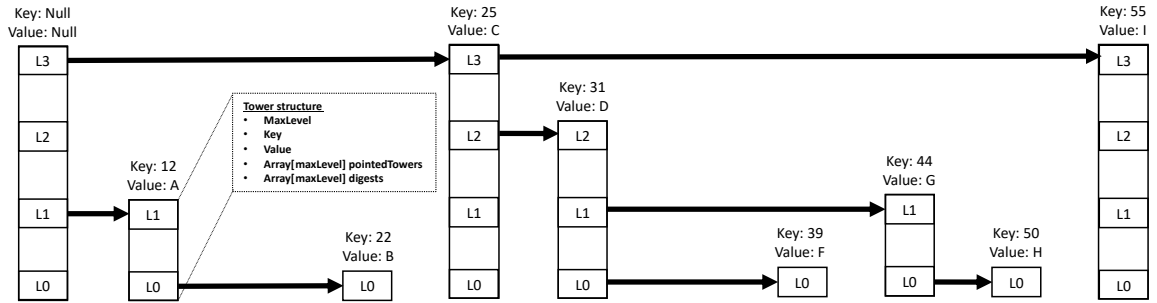


Figure 8.2: AMVSL - skip list structure

versioned row values. In the following sections we present the details regarding AMVSL and its composing data structures. We first discuss the index's skip list structure, which manages the keys.

8.2.1 Index structure overview

Fig. 8.2 shows an example of an authenticated multi-version skip list (AMVSL) index. The AMVSL structure is composed of multiple nodes, where each node is connected to its immediate vertical or horizontal neighboring nodes. We refer to vertically connected nodes as a **tower** where the first/left-most tower is referred to as **head tower**. Each tower represents a key-value entry and has a certain height, e.g., the second tower from the left is of height 1 (zero-based), which represents key 12 with value A. The tower's height is equal to the number of the tower's comprising nodes. We refer to each tower's node in a specific level as a **tower level**. The main difference in our index structure, compared to the classic skip list structure is that each tower can be visited only through its tower's topmost level (or max level), from a pointing tower with an equal or higher max level. This property limits the flow connectivity to be only between two sequential towers with equal or descending max levels. This creates unique traversal paths starting from the head tower (first tower to the left containing NULL key and value), and ending in terminal towers (leaves) that do not point to any tower. This effectively converts the randomized skip list

Partition 2						
	Digest	ValidFrom	ValidTo	Col1	...	ColN
Partition 1	Digest	ValidFrom	ValidTo	Col1	...	ColN
	Digest	ValidFrom	ValidTo	Col1	...	ColN
Partition 0	Digest	ValidFrom	ValidTo	Col1	...	ColN
	Digest	ValidFrom	ValidTo	Col1	...	ColN

Figure 8.3: AMVSL - bucket structure (partition capacity of 2)

structure to a randomized B^+Tree like structure, where the max size of each branch is randomly generated.

The index's root digest is computed similarly as in Merkle tree like data structures. Since each branch contains unique traversal path for each key, the authentication process is done exclusively on the traversal path, as we further discuss in Section 8.2.6. An additional difference in the AMVSL structure, compared to the classic skip list structure is that every node's vertical neighbor is logically and physically grouped into a **tower**. The tower contains the associated key, value, and two aligned arrays: an array containing the references to the tower's pointed towers at each level and an additional array of hash digests that stores intermediate verification values. By using arrays, AMVSL enables accessing each tower's vertical level using its position which eliminates the need for pointer chasing when traversing the tower's nodes vertically.

8.2.2 Versioning bucket data structure overview

To help support data versioning, AMVSL additionally incorporates a versioning bucket data structure. Each tower contains a single bucket, which manages its versioned values (rows). Fig. 8.3 presents the logical structure of the bucket data structure. The bucket is composed of row entries, which contain the columns according to the table schema in addition to the following three fields:

- **digest** - verification digest encapsulating a bucket's row and its preceding rows
- **validFrom** - version from which the row entry is valid (it is also the version when

the row entry was created)

- **validTo** - version in which the row entry is no longer valid (it is also the version when the corresponding row was changed or deleted). It is initialized to ∞ . This defines the row entry's validity range as $[validFrom, validTo)$, i.e., it is valid from version *validFrom* (including) up to and not including version *validTo*.

As the index versions are created sequentially in a strictly ascending order, the added bucket's row entries are inherently sorted in a descending order with the latest version in the top row entry. Since a key's latest row version is at the top of the bucket it can be accessed directly. This can restrict the index search space to row keys only, when a query is designated to work on latest data (as in authenticated single-version indexes).

To illustrate how the AMVSL index works we use a simple scenario, where the table schema contains a key and value columns denoted as (key,value). The relational table is created with an initial version of 0. Three rows are inserted: (100, 1000), (200, 2000), (300, 3000) (see Figure 8.4(a)). In each row entry **validFrom** contains the version in which it was created and **validTo** is set to ∞ . Assume that *in the same version*, 100 is added to all rows' values (Fig. 8.4(b)), i.e., each row entry is changed without modifying their corresponding **validFrom** or **validTo** fields. We next commit the current index version to preserve current table's snapshot, thereby advancing the current table's version to 1.

We next perform the following operations: delete key 300, update key 200's value to 2220, and insert a new entry (400, 4000) (Fig. 8.4(c)). To encode that key 200's deleted row entry is valid until the current version (not including) its **validTo** is set to current version 1; Updating key 200's value to 2220, preserves the previously committed value (**validTo** is set to version 1) and creates a new row entry where **validFrom** is set to 1, which indicates that the current value (2220) is valid from version 1 (including). Inserting (400, 4000) creates a new skip list tower with its

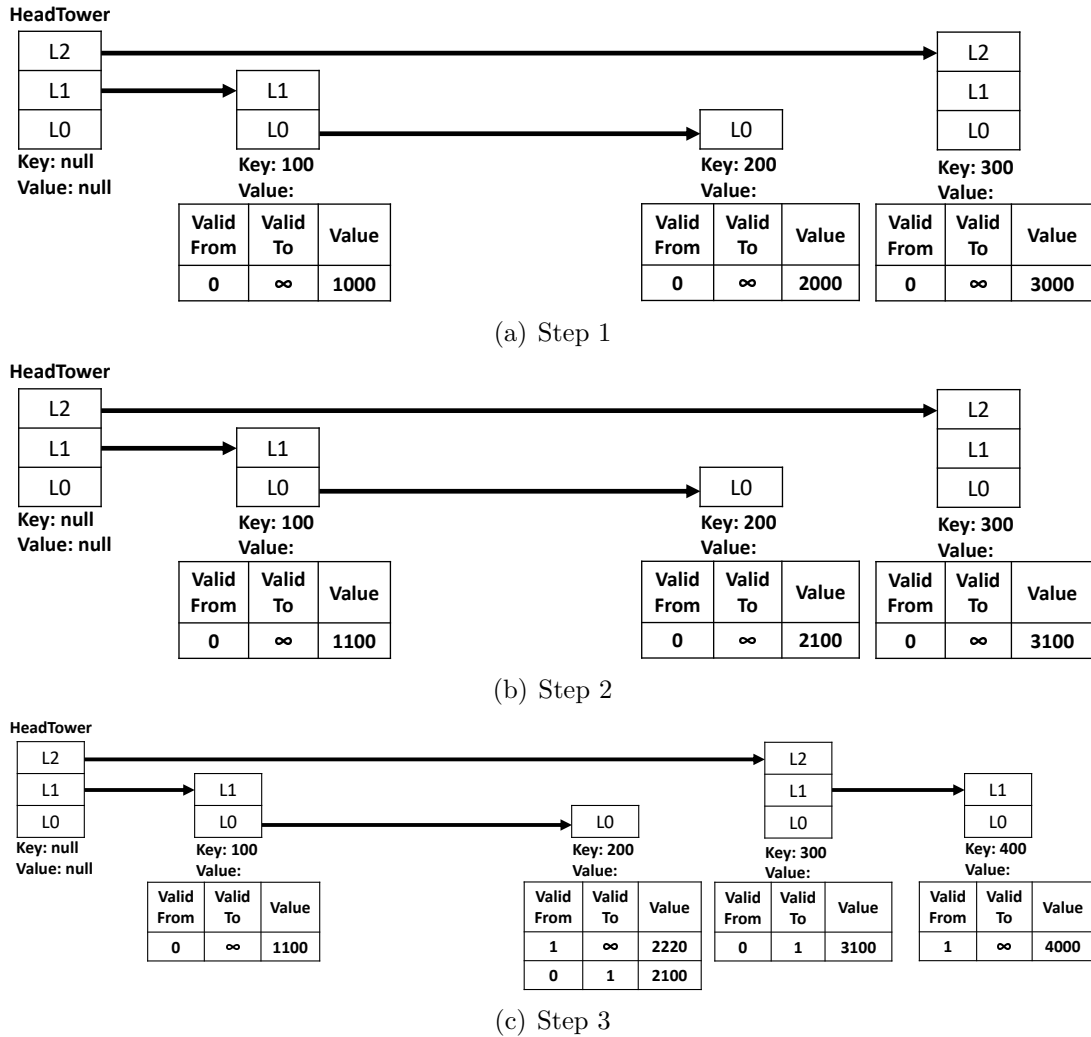


Figure 8.4: An example of AMVSL index

corresponding bucket row entry. We next discuss the AMVSL upsert and delete algorithms in detail.

8.2.3 Upsertion algorithm

In an upsert operation, a new key is added to the index only if the index does not already contain that key. If the index contains such a key, the index key's value is updated with the new key's value.

We continue with the current example (Fig. 8.4(c)) and insert a new row (150,1500) to the AMVSL index. The insertion of a new row depends on whether or not a

tower with the inserted row's key already exists. This can occur if a previous version contained committed rows with the same key, which were not modified. If such a tower exists, the new row can be added to the top of the existing tower. If no such tower exists, a new tower is inserted to accommodate the new row. Due to the index property where a pointer to a tower can only be created at the pointed tower's max level, a new tower can only be inserted when the current traversed level equals the new tower's max level.

When inserting the row with key 150, a new tower is created with a randomly generated max level of 1 (zero-based). The algorithm starts its traversal from the head tower at its max level (level 2 in our example). The pointed tower's key (300) is compared to the new tower's key (150). Since the pointed tower's key is higher, this would be the place to add the new tower since its key precedes the pointed tower's key, however since the max level of the new tower (level 1) is lower than the current traversed level it is not possible. The search for a new insert position continues in the head tower from the next lower level (level 1). As the current pointed tower's key (100) is lower than the new tower's key, the search continues from the pointed tower at the same level (level 1). Since the current tower does not point to any tower and the current level 1 equals the new tower's max level, this point is chosen as the potential insert position. Note that it is possible for an existing tower with the same key as the new tower's key to exist in the yet to be visited lower levels of the index. The final part of the new tower's insertion depends on whether such an existing tower is encountered. The traversal continues at the tower with key 100 on the next lower level (level 0). Since the pointed tower's key (200) is higher than the new tower's key (150) the next lower level should be traversed. As there are no more levels to traverse, the algorithm concludes that the new tower should be inserted and backtracks its path to finalize the insert. The new tower is inserted between the towers with keys 100 and 200. The pointer from the tower with key 100 to tower

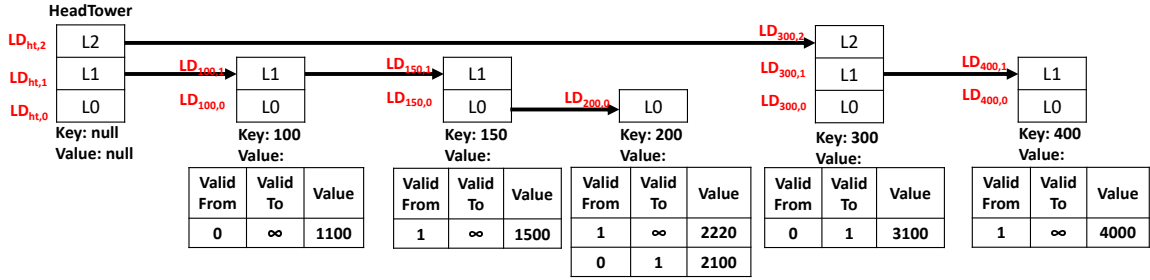


Figure 8.5: AMVSL index scenario after inserting key 150

with key 200 at level 0 is removed and a new pointer at the same level is added to the new tower, which now points to the tower with key 200. Next, the tower with key 100 at level 1 is set to point to the new tower’s max level. Fig. 8.5 shows the index after inserting the new tower with the new row (150,1500).

The above example illustrates the insertion of a new row in a new tower. The insertion algorithm, which involves the insertion of a row to an existing tower with the same key (e.g., on update), is similar with the exception that when the existing tower with the same key is found, no pointer changes are applied when backtracking the path. We next describe Algorithm 5: Upsert in more detail. The row is opti-

Algorithm 5: Upsert

input : row
1 $newTowerMaxLevel \leftarrow randomMaxLevel$;
2 $headTower.extendHeadTowerMaxLevel(newTowerMaxLevel)$;
3 $upsertInner(headTower.maxLevel, headTower, row.key, row, newTowerMaxLevel)$;

mistically assumed to be inserted into a new tower. The new tower’s max level is randomly generated (line 1), which is bounded by a predefined number (representing the maximum allowed tower level, e.g., 20 in our experiments). If the new tower has a max level, which is larger than that of the head tower, the head tower’s max level is expanded to match the new tower’s max level (line 2). The algorithm next calls the recursive Algorithm 6: UpsertInner.

Algorithm 6: UpsertInner traverses the index towers’ levels, starting from the head tower at its max level and compares the new tower’s key to the pointed tower’s key at the same level. If the latter is smaller than the new tower’s key (lines 35-36), the

Algorithm 6: UpsertInner

```
input : level, tower, key, row, newTowerMaxLevel
1 if level  $\geq$  0  $\wedge$  tower  $\neq$  NULL then
2   pointedTower  $\leftarrow$  tower.pointedTower(level);
3   if pointedTower = NULL then
4     if level = newTowerMaxLevel then
5       newTower  $\leftarrow$  newTower(currentVersion, row, newTowerMaxLevel, partitionCapacity);
6       existingTower  $\leftarrow$  processLowerLevelsForUpsert(level - 1, tower, newTower);
7       if existingTower = NULL then
8         processTowerDigests(level, newTower);
9         tower.setPointedTower(level, newTower);
10        tower.processLevelDigest(level, newTower);
11      else
12        tower.processLevelDigest(level, NULL);
13    else
14      upsertInner(level - 1, tower, key, row, newTowerMaxLevel);
15      tower.processLevelDigest(level, NULL);
16  else
17    if pointedTower.key = key then
18      updateTower(row, pointedTower);
19      tower.processLevelDigest(level, pointedTower);
20    else if pointedTower.key > key then
21      if level = newTowerMaxLevel then
22        newTower  $\leftarrow$  newTower(currentVersion, row,
23          newTowerMaxLevel, partitionCapacity);
24        existingTower  $\leftarrow$  processLowerLevelsForUpsert(level - 1, tower, newTower);
25        if existingTower = NULL then
26          newTower.setPointedTower(level, pointedTower);
27          processTowerDigests(level, newTower);
28          tower.setPointedTower(level, newTower);
29          tower.processLevelDigest(level, newTower);
30        else
31          tower.processLevelDigest(level, pointedTower);
32      else
33        upsertInner(level - 1, tower, key, row, newTowerMaxLevel);
34        tower.processLevelDigest(level, pointedTower);
35    else
36      upsertInner(level, pointedTower, key, row, newTowerMaxLevel);
37      tower.processLevelDigest(level, pointedTower);
```

search continues from the pointed tower at the same level. If the current level is bigger than the new tower's max level and the current pointed tower's key is larger than the new tower's key (lines 32-33) or is NULL (lines 14-15), the search continues on the current tower on the following lower level. Note that each tower is constrained by its maximum level, i.e., it can only be entered from its max level; hence, a new tower's candidate insert position can only be found where the current level equals the new tower's max level and either: (1) the current tower does not point to a tower at this level (lines 3-12) or (2) the current tower points to a tower, which

contains a key that is bigger than the new tower's key (lines 20-30). Since the insert candidate position is determined by the randomly generated new tower's max level, it is possible that a tower with the same key as the new tower's key already exists in the yet to be visited lower levels of the index. Subsequently, the algorithm continues searching for such an existing tower (Algorithm 7: ProcessLowerLevelsForUpsert is called on lines 6 and 23). The search for the new tower's key ends when either an existing tower with the same key is found (lines 17-19) or when all the levels are traversed from top to bottom. If the new tower is inserted successfully, i.e., no other existing tower has the same key as the new tower, a pointer to the new tower's max level is added in the case the new tower is added at the end of the search path (lines 7-10), and in the case the new tower is added between two existing towers, the pointers of the two existing towers and the new tower are arranged accordingly at the new tower's max level (lines 24-28).

Algorithm 7: ProcessLowerLevelsForUpsert

```

input : level, tower, newTower
output: existingTower
1 existingTower  $\leftarrow$  NULL;
2 if  $level \geq 0 \wedge tower \neq NULL$  then
3   pointedTower  $\leftarrow$  tower.pointedTower(level);
4   if pointedTower = NULL then
5     existingTower  $\leftarrow$  processLowerLevelsForUpsert(level - 1, tower, newTower);
6     tower.processLevelDigest(level, NULL);
7   else
8     if pointedTower.key < newTower.key then
9       existingTower  $\leftarrow$  processLowerLevelsForUpsert(level, pointedTower, newTower);
10      tower.processLevelDigest(level, pointedTower);
11     else if pointedTower.key > newTower.key then
12       existingTower  $\leftarrow$  processLowerLevelsForUpsert(level - 1, tower, newTower);
13       if existingTower = NULL then
14         newTower.setPointedTower(level, pointedTower);
15         tower.setPointedTower(level, NULL);
16         tower.processLevelDigest(level, NULL);
17       else
18         tower.processLevelDigest(level, pointedTower);
19     else
20       existingTower  $\leftarrow$  pointedTower;
21       row  $\leftarrow$  newTower.value.lastRow;
22       updateTower(row, existingTower);
23       tower.processLevelDigest(level, pointedTower);
24 return existingTower;

```

Algorithm 7: `ProcessLowerLevelsForUpsert` is called after a candidate insertion position was found for the new tower. It traverses the index in the sub range that is determined by the keys of `tower` and `newTower`, starting from the current tower at level `newTower.maxLevel - 1`. The new tower's key is compared to each traversed pointed tower key. If the current pointed tower's key is NULL, the search continues on the same current tower on the following lower level (lines 4-6). If the pointed tower is smaller than the new tower's key (lines 8-10), the search continues from the pointed tower at the same level. When the current pointed tower's key is bigger than the new tower's key (lines 11-12), the new tower will potentially be placed between the current tower and the pointed tower. At this point the pointer from the current tower is removed (line 15) and is added to the new tower at the same level (line 14) and the search continues on the same current tower on the following lower level. If the current pointed tower's key equals the new tower key, the new tower should not be added and the insertion of the new row should be done on the existing tower (lines 19-23). At this point the tower, which contains the existing key is stored (line 20) to be later returned recursively, the new row is extracted from the new tower (line 21) and is attempted to be added to the existing tower's bucket (line 22). Due to the recursive structure of the algorithm, the pointer modifications are done after the entire sub index was traversed. At this point it is already known if an existing tower with the same key as the new tower was found. Hence, when the algorithm backtracks, the pointers modification occurs only if the new tower was inserted, i.e., the returned existing tower is NULL (lines 13-16).

Algorithm 8: `UpdateTower`

```

input : row, towerToUpdate
1 lastRow ← towerToUpdate.value.lastRow;
2 lastRowVersion ← lastRow.version;
3 if lastRowVersion.validTo = ∞ then
4   | towerToUpdate.value.updateLastRow(currentVersion, row);
5 else
6   | towerToUpdate.value.add(currentVersion, row);
7 processTowerDigests(towerToUpdate.maxLevel, towerToUpdate);

```

To update a tower with an existing key, Algorithm 6: `UpsertInner` and Algorithm 7: `ProcessLowerLevelsForUpsert` call Algorithm 8: `UpdateTower`. In this algorithm, if in the bucket's top row $validTo \neq \infty$ (encoding that the row was committed in a previous version and that the current version does not contain a row with the same key), the row is inserted to the top of the bucket with $validFrom$ set to $currentVersion$ and $validTo$ set to ∞ (abstracted by the call to `add` in line 6). In the case where in the bucket's top row's $validTo = \infty$ (encoding that the current version contains a row with the same key), `updateLastRow` (line 4) is called, which abstracts the following logic: if $validFrom = currentVersion$, the existing row entry's value is replaced with the provided row's value. Alternatively, if $validFrom < currentVersion$ this means that the current existing row entry's value was committed and needs to be preserved. To achieve this the current existing row entry's $validTo$ is set to $currentVersion$ and the new row is inserted with $validFrom$ set to $currentVersion$ and $validTo$ set to ∞ .

8.2.4 Deletion algorithm

The `Delete` algorithm traverses the index until a tower with the required key is found. If the found tower bucket's top row was added in a version that was not yet committed, that row is removed from the bucket. If the bucket is empty as a result, the entire tower is removed by rearranging the pointers of the neighboring towers in contrast to the way they are rearranged in Algorithm 7: `ProcessLowerLevelsForUpsert`. We next describe the `Delete` algorithm in more detail.

The recursive Algorithm 9: `DeleteInner` is called by the `Delete` algorithm. It traverses the index similar to Algorithm 6: `UpsertInner`. When a tower containing the key to delete is found (line 9) Algorithm 10: `DeleteTower` is called to delete the row if it exists in the current version (`currentVersion`).

Algorithm 10: `DeleteTower` starts by attempting to delete the current version's row

Algorithm 9: DeleteInner

```
input : level, tower, key
output: deletedRow
1 deletedRow  $\leftarrow$  NULL;
2 if level  $\geq$  0  $\wedge$  tower  $\neq$  NULL then
3   pointedTower  $\leftarrow$  tower.getPointedTower(level);
4   if pointedTower = NULL then
5     deletedRow = deleteInner(level - 1, tower, key);
6     if deletedRow  $\neq$  NULL then
7       tower.processLevelDigest(level, pointedTower);
8   else
9     if pointedTower.key = key then
10      deletedRow  $\leftarrow$  deleteTower(level, tower, pointedTower);
11      if deletedRow  $\neq$  NULL then
12        deletedTowerPointedTower  $\leftarrow$  pointedTower.getPointedTower(level);
13        tower.processLevelDigest(level, deletedTowerPointedTower);
14      else if pointedTower.key > key then
15        if level > 0 then
16          deletedRow  $\leftarrow$  deleteInner(level - 1, tower, key);
17          if deletedRow  $\neq$  NULL then
18            tower.processLevelDigest(level, pointedTower);
19        else
20          deletedRow  $\leftarrow$  deleteInner(level, pointedTower, key);
21          if deletedRow  $\neq$  NULL then
22            tower.processLevelDigest(level, pointedTower);
23 return deletedRow;
```

(topmost bucket row) (line 1). If the row is valid in the current version (encoded by $validTo = \infty$) `deleteLastRow` is called, which abstracts the following logic: if $validFrom = currentVersion$, the existing row entry's is removed. Alternatively, if $validFrom < currentVersion$ this means that the current existing row entry's value was committed and needs to be preserved. To achieve this the current existing row entry's $validTo$ is set to $currentVersion$. The algorithm continues to check if the resulting bucket is empty. If it is empty (lines 2-6) Algorithm 11: `ProcessLowerTowerLevelsForDelete` is called to first rearrange the lower level pointers to bypass the deleted tower (line 3) and continue to rearrange the current tower level to bypass the delete tower.

Algorithm 11: `ProcessLowerTowerLevelsForDelete` traverses the lower levels of the index that are bound by the `towerToDelete`. If the current traversed tower level does not point to any tower (line 3), the algorithm sets the current traversed tower

Algorithm 10: DeleteTower

```
input : level, tower, towerToDelete
output: deletedRow
1 deletedRow  $\leftarrow$  towerToDelete.value.deleteLastRow(currentVersion);
2 if towerToDelete.value.isEmpty then
3   | processLowerTowerLevelsForDelete(towerToDelete.maxLevel - 1, tower, towerToDelete);
4   | pointedPointedTower  $\leftarrow$  towerToDelete.getPointedTower(level);
5   | tower.setPointedTower(level, pointedPointedTower);
6   | tower.processLevelDigest(level, pointedPointedTower);
7 else
8   | processTowerDigests(towerToDelete.maxLevel, towerToDelete);
9   | tower.processLevelDigest(level, towerToDelete);
10 return deletedRow;
```

Algorithm 11: ProcessLowerTowerLevelsForDelete

```
input : level, tower, towerToDelete
1 if level  $\geq$  0  $\wedge$  tower  $\neq$  NULL then
2   | pointedTower  $\leftarrow$  tower.getPointedTower(level);
3   | if pointedTower = NULL then
4     | pointedPointedTower  $\leftarrow$  towerToDelete.getPointedTower(level);
5     | processLowerTowerLevelsForDelete(level - 1, tower, towerToDelete);
6     | tower.setPointedTower(level, pointedPointedTower);
7     | tower.processLevelDigest(level, pointedPointedTower);
8   | else
9     | processLowerTowerLevelsForDelete(level, pointedTower, towerToDelete);
10    | tower.setPointedTower(level, pointedTower);
11    | tower.processLevelDigest(level, pointedTower);
```

level to point to the `towerToDelete` pointed tower at the same level in order to bypass the `towerToDelete` (lines 4,6).

8.2.5 Index commit

Committing the current index version snapshot consists of assigning the current index version with a strictly higher version than the current index version. The preservation of committed row entries is enforced by the upsert and delete algorithms as discussed above.

8.2.6 Index authentication process

In this section we discuss the authentication process in the two index's data structures, i.e., skip list and bucket. We start with detailing the bucket verification digest computation that is used in the skip list verification digest computation.

Skip list bucket verification digest computation As shown in Fig. 8.3, each tower’s bucket row entry contains a digest field. This field is computed using a cryptographic hash function (e.g., SHA256) on all the rows’ fields (i.e., `validFrom`, `validTo`, `col1`, ..., `colN`) in a predefined order, and on the digest value of the previous row entry (if one exists) or with a NULL digest, otherwise. Using this “onion layered” digest computation approach, each bucket row entry digest contains its own fields’ digest in addition to all previous row entries’ digests. Particularly, the bucket’s topmost row entry’s digest encapsulates all of the bucket row entries’ digests. The digest is computed when a row entry is added to the bucket and is recomputed when any row entry’s column is updated, i.e., on updating the row’s data columns or the `validTo` field. When a row is deleted from a bucket, i.e., when the bucket’s row was added in an uncommitted version and is then deleted in the same uncommitted version, the entire row and its digest are safely removed. The previous bucket row entry’s digest (if the bucket is not empty) is composed of its own fields and the preceding row’s digest and does not need to be recomputed.

Skip list tower level verification digest computation In the following we discuss the verification digest computation performed in Algorithm 6: `UpsertInner`, the verification digest computation in Algorithm 9: `DeleteInner` is performed similarly. Algorithm 6: `UpsertInner` performs the traversal of the index recursively, which reaches its end when either: (1) the new row is upserted within an existing tower with the same row’s key or (2) the new tower, which contains the new row is inserted. After either scenario the algorithm backtracks the traversed towers’ levels and for each such tower level it computes and stores the corresponding digest. This is done by the tower’s call to `processLevelDigest` in Algorithm 6: `UpsertInner` and Algorithm 7: `ProcessLowerLevelsForUpsert`, which takes two arguments: current level and pointed tower. The digest is computed differently for towers’ level 0 and higher levels. For level 0 the digest is computed from the tower’s key, tower bucket’s topmost entry

row digest, and the digest of the pointed tower level if it exists or a NULL digest otherwise. For each tower level higher than 0, the digest is computed from the tower's immediate lower level and the pointed tower level digests.

The AMVSL index's state after inserting key 150 is shown in Fig. 8.5. A red label in the form of $LD_{key,level}$ represents each tower's level digest, which is stored in its digests array, e.g., $LD_{200,0}$ represents the level digest (LD) of the tower with key 200 in level 0. We define an entire bucket's digest (BD) as the digest of its top-most row entry and represent it with the label BD_{key} , e.g., the bucket digest of the tower with key 200 is represented by BD_{200} . After the new tower with key 150 is added, its zero level digest $LD_{150,0}$ is computed from its key (150), its value BD_{150} , and the digest of the pointed tower level at level 0 ($LD_{200,0}$). Hence, $LD_{150,0} = H(150, BD_{150}, LD_{200,0})$, where H denotes the cryptographic hash function, which can operate on multiple parameters. The digest of $LD_{150,1}$ is computed from its bottom level digest ($LD_{150,0}$) and its pointed tower level digest at level 1. Since no such tower level exist, the $H(NULL)$ digest is used; hence, $LD_{150,1} = H(LD_{150,0}, H(NULL))$, similarly the following digests are computed as follows: $LD_{100,1} = H(LD_{100,0}, LD_{150,1})$, $LD_{ht,1} = H(LD_{ht,0}, LD_{100,1})$, where $LD_{ht,1}$ represents the digest of the head tower in level 1. Finally $LD_{ht,2} = H(LD_{ht,1}, LD_{300,2})$, where $LD_{ht,2}$ encapsulates the digest of the head tower's max level and hence of the entire AMVSL.

8.2.7 Query processing algorithms

Our proposed AMVSL structure is designed to efficiently support advanced query capabilities, i.e., SVRK, MVRK and MVAK, on historical blockchain data. We next discuss the algorithm to search for a range of keys across a range of versions, which when provided with different parameters can enable the different search capabilities. For example, to search for a single key in a single version, the range of keys will contain the same single key as the start and end key and the range of versions will

contain the single version as the start and end versions.

The AMVSL index uses the range search capabilities of two main data structures: the skip list-like data structure is used to search for keys in the keys range. For each located key, the corresponding tower bucket’s data structure is searched for versions in the versions range.

Algorithm 12: RangeSearch

input : verStart, verEnd, keyStart, keyEnd, foundRows
1 *rangeSearchInner*(*headTower.maxLevel*, *verStart*,
verEnd, *headTower*, *keyStart*, *keyEnd*, *foundRows*);

Algorithm 13: RangeSearchInner

input : level, verStart, verEnd, tower, keyStart, keyEnd, foundRows
1 **while** *level* ≥ 0 **do**
2 *pointedTower* ← *tower.pointedTower*(*level*);
3 **if** *pointedTower* = NULL **then**
4 *level* --;
5 **else**
6 **if** *pointedTower.key* ≥ *keyStart* **then**
7 **if** *pointedTower.key* ≤ *keyEnd* **then**
8 *rangeSearchInner*(*level* - 1, *verStart*, *verEnd*
9 , *tower*, *keyStart*, *keyEnd*, *foundRows*);
10 *pointedTower.value.search*(*verStart*, *verEnd*, *foundRows*);
11 *tower* ← *pointedTower*;
12 **else**
13 *level* --;
14 **else**
15 *tower* ← *pointedTower*;

Multi-version skip list range search Algorithm 12: RangeSearch presents the general AMVSL range search that consists of a call to the recursive Algorithm 13: RangeSearchInner.

Algorithm 13 traverses the index starting from the head tower at its max level (as provided by the caller). The search for the required keys’ range continues to follow the following guidelines (as long as the traversed tower level is non negative): if the current tower level does not point to any tower, the search continues on the current tower’s next lower level (lines 3-4). If the current pointed tower’s key is smaller than *keyStart*, the search continues from the pointed tower at the same level (line 14).

Otherwise, the current pointed tower's key is bigger or equal to `keyStart` (line 6). If it is also bigger than `keyEnd`, it is not in the required range and we continue to search the index from the next lower level (line 12). When the current pointed tower's key is also smaller or equal to `keyEnd` (line 7) implies that the current pointed tower's key is in the required key range and its bucket rows are searched for the requested version range (line 9), which will be discussed shortly. From this point the search continues in two branches: (1) from next pointed tower (line 10) and (2) in the index's sub range between the current tower and its pointed tower keys; This is due to the fact that in the index, for any subsequent towers, i.e., *towerA* and *towerB*, all inner tower keys maintain $towerA.key < innerTower.key < towerB.key$. The search recursively continues to extract all keys from the index sub range (line 8) before continuing to extract the current pointed towers' valid bucket rows. This enables retrieving the resulting rows in a sorted manner.

Bucket versions range search When a tower with a specific key is located its row entries are located in the tower's contained bucket data structure. As the bucket's row entries are inherently sorted on the `validFrom` field, a binary search can be performed to efficiently locate a specific version. To enable a more efficient search, we reduce the search space by dividing each bucket into partitions of predefined size (e.g., capacity of 2 in Fig. 8.3). Since each partition contains sorted row entries, a version binary search can be performed on the partitions' `validFrom` range, which is extracted from its first and last row entries. We next discuss the algorithms for efficiently searching a bucket's versions range across partitions.

Algorithm 14: `PartitionsSearch` performs a search on the bucket's partitions for valid row entries in the range `[verStart, verEnd]`. The found valid rows are stored in `foundRows`.

The algorithm determines the position of the first bucket row, which contains the largest `validFrom`, which is smaller or equal to `verStart` (line 3) by calling Al-

Algorithm 14: PartitionsSearch

```
input : verStart, verEnd, foundRows
1 if partitions.isEmpty then
2   return;
3 pos ← posRowWithVersionEqualSmall(verStart);
4 partitionPos ← pos/partitionCapacity;
5 rowPos ← pos%partitionCapacity;
6 while partitionPos < partitions.size do
7   partition ← partitions[partitionPos];
8   while rowPos < partition.size do
9     runningRow ← partition[rowPos];
10    rowVersion ← runningRow.version;
11    if rowVersion.validFrom > verEnd then
12      return;
13    if rowVersion.validTo = ∞ ∨ rowVersion.validTo > verStart then
14      foundRows.add(runningRow);
15    rowPos ++;
16 rowPos ← 0;
17 partitionPos ++;
```

gorithm 15: PosRowWithVersionEqualSmall, which will be discussed shortly. This position is then converted to the row's containing partition position (line 4) and to the row position in that partition (line 5). From this row position in its containing partition all consecutive rows in all consecutive partitions are sequentially examined and the valid rows in the requested version range are extracted. This sequential search ends when either the current examined row was created after **verEnd** (lines 11-12) or until all rows are exhausted. We next discuss the details of the called Algorithm 15: PosRowWithVersionEqualSmall.

Algorithm 15: PosRowWithVersionEqualSmall returns the absolute position of the bucket containing the first row that has a **validFrom**, which is equal to or smaller than the requested version.

If the bucket is empty, no such row exists, the algorithm returns -1 (lines 1-2). The algorithm continues to perform a binary search for the first partition that contains the required row position (lines 3-14). The comparison of a partition to a version (lines 9,11) is defined as follows: $partition = version$ iff $partition.firstRowValidFrom \leq version \leq partition.lastRowValidFrom$; $partition < version$ iff $partition.lastRowValidFrom < version$; and

Algorithm 15: PosRowWithVersionEqualSmall

```
input : version
output: pos
1 if partitions.size = 0 then
2   return -1;
3 start ← 0;
4 end ← partitions.size - 1;
5 mid ← -1;
6 while start ≤ end do
7   mid ← start + ((end - start)/2);
8   partition ← partitions[mid];
9   if partition = version then
10    break;
11  else if partition < version then
12    start ← mid + 1;
13  else
14    end ← mid - 1;
15 pos ← mid;
16 if start ≤ end then
17   partition ← partitions[pos];
18   rowPos ← partition.posEqualOrSmallerPos(version);
19   pos ← (pos * partitionCapacity) + rowPos;
20 else if start = partitions.size then
21   pos ← elemCount - 1;
22 else if end < 0 then
23   pos ← 0;
24 else
25   partition ← partitions[end];
26   pos ← (end * partitionCapacity) + partition.size - 1;
27 return pos;
```

$partition > version$ iff $partition.firstRowValidFrom > version$.

If the partition was found (line 16), an additional binary search is performed inside that partition (lines 17-18) for the position of the row entry with the largest `validFrom` that is equal or smaller than the required version. This row's position and its partition's position are then converted to an absolute bucket row position (line 19). If all bucket's rows were created before the requested version, the position of the bucket's last row is returned (line 20-21). Alternatively, if all bucket's rows were created after the requested version, the position of the bucket's first row is returned (lines 22-23). In the last possible scenario, where the requested version was found between two partitions, the `end` parameter contains the position of the partition with the largest `validFrom` that is smaller than the requested version in its last row entry. That row entry position is used as the returned row position (lines

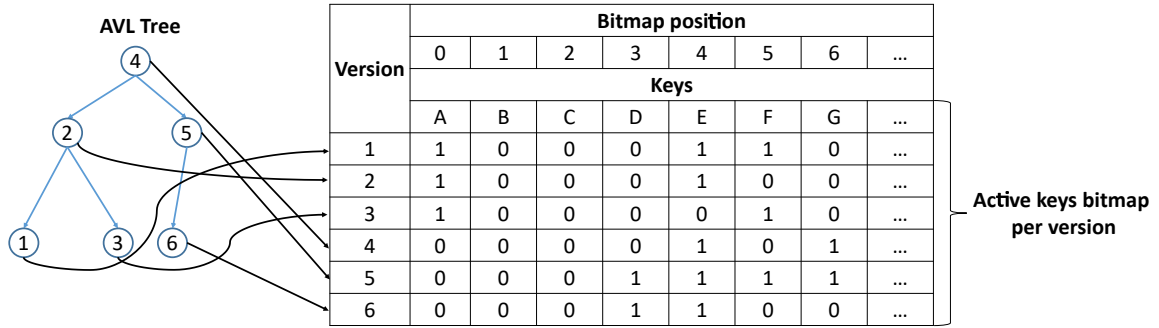


Figure 8.6: AMVSL - versions to active keys data structure

25-26).

8.2.8 Efficiently querying all keys across a range of versions

The AMVSL index enables an efficient search of a single key in a specific version, a single key across a range of versions, a range of keys in a specific version or a range of keys across a range of versions. This is done by first traversing the index's skip list to the tower containing the required key and then searching the tower's bucket for the versions' range. When it is required to search all valid rows in a range of versions, the index needs to traverse all towers and search each tower's bucket for the valid rows. The scenarios requiring this functionality are ubiquitous and usually involve a running key, e.g., in an anti-money laundering system that flags irregular funds' transfer between account numbers (running keys) in specific dates (versions), a common query would require retrieving all suspected account numbers that were flagged at a specific time range (e.g., last three months). To support such scenarios we designed a data structure (VersionsToKeys), which efficiently maps each version to its active keys.

Fig. 8.6 shows the overview of the VersionsToKeys data structure. All (unique) running keys are mapped to their unique positions. The mapping is implemented using a keys array where each key is mapped to its zero-based position, e.g., key A is mapped to position 0, and key C to position 2. Each version is mapped to a bitmap,

where its bits' positions correspond to the running keys positions. Bits that are set to 1 represent the keys that contain valid rows in the mapped version. The mapping between the versions to their corresponding bitmap is implemented using an AVL tree index. Since many scenarios do not contain most keys, their compressed bitmap representation requires minimal storage consumption. To support this, we use the RoaringBitmap java package [89], which enables efficient operations on a compressed bitmap.

When the AMVSL index adds a key, the key's corresponding bitmap position is set to 1. When a key is removed from the current version (i.e., when deleting a key) the corresponding bitmap position is set to 0. The resulting bitmap contains 1 only for keys that are active in the current version.

When a range of versions is queried, the AVL tree index is traversed to retrieve the bitmaps of the corresponding versions range. A logical OR operation is performed on the retrieved bitmaps, which results in a single bitmap representing all valid keys in the requested versions range. The resulting bitmap is then converted to a set of keys by querying the keys array for each set bit's position. The AMVSL index then queries the versions range for each key in the resulting key set.

We illustrate the approach in the anti-money laundering example scenario. Assume that accounts A, E and F were flagged on day 1 (first row alongside version 1 in Fig. 8.6), on the next day accounts A, E were flagged, and on day 3 accounts A and F were flagged. To retrieve information of only flagged accounts between days (versions) 1 and 3, the AVL tree index is traversed, the bitmaps for these days range are extracted, on which a logical OR operation is performed, i.e., $1000110 \vee 1000100 \vee 1000010 = 1000110$. A set of keys with the position of each set bit in the resulting bit map is extracted, i.e., A, E, F. The AMVSL index then searches for each key (account) in the required version range to provide the full account information. We next describe the VersionsToKeys data structure algorithms for insert, delete,

commit, and versions range search.

In the following algorithms `uncommittedBitmap` contains a (roaring) bitmap of all keys in the current (uncommitted) version, `keys` contains an array of all existing unique keys in a sorted order (this is the mapping of all keys to their array zero-based position, which in turn corresponds to their bitmap positions). `versionsToBitmap` contains the AVL tree index, which stores the mapping of all versions to their corresponding bitmap, and `currentVersion` contains the current uncommitted version identifier.

Algorithm 16: VersionsToKeyAdd

```
input : key
1 pos ← binarySearch(keys, key);
2 if pos ≠ NULL then
3   | uncommittedBitmap.add(pos);
4 else
5   | keys.add(key);
6   | uncommittedBitmap.add(keys.size - 1);
```

Algorithm 16: VersionsToKeyAdd adds an existing or new key in the current version to the VersionsToKeys data structure.

Initially the added key is searched for in the existing keys array using a binary search (line 1). If the key already exists its position is retrieved and is used to set the current bitmap in that position to 1 (lines 2-3). If no position is found, the key is appended at the end of the keys array (line 5) and bit 1 is appended to the end of `uncommittedBitmap`, where the added position is the added key's position in the `keys` array (line 6).

Algorithm 17: VersionsToKeysDelete

```
input : key
1 pos ← binarySearch(keys, key);
2 if pos ≠ NULL then
3   | uncommittedBitmap.remove(pos);
```

Algorithm 17: VersionsToKeysDelete deletes an existing key from the VersionsToKeys data structure.

The key is searched in the existing `keys` array using a binary search (line 1). If the key was found, its position is retrieved and is used to unset the current bitmap position (lines 2-3). Note that when a row with an existing valid key is updated there is no need to update the VersionsToKeys data structure since the key is still valid in the current version.

Algorithm 18: VersionsToKeysCommit

```

input : nextVersion
1 uncommittedBitmap ← uncommittedBitmap.clone;
2 currentVersion ← nextVersion;
3 versionsToBitmap.insert(currentVersion, uncommittedBitmap);

```

Algorithm 18: VersionsToKeysCommit commits the current version’s VersionsToKeys data structure.

A copy of the current uncommitted bitmap is saved to keep track of all key changes in the next version (line 1). The next provided version (`nextVersion`) is set to be the current version to track (line 2). A reference to the current `uncommittedBitmap` is also stored in the `versionsToBitmap`’s AVL tree index (line 3) to accommodate the operations in the next uncommitted version.

Algorithm 19: VersionsToKeysGetKeys

```

input : verStart, verEnd
output: keys
1 bitmap ← RoaringBitmap;
2 for curBitmap ∈ versionsToBitmap.values(verStart, verEnd) do
3   | bitmap.or(curBitmap);
4 keysRet ← HashSet;
5 for setBitPos ∈ bitmap do
6   | keysRet.add(keys[setBitPos]);
7 return keysRet;

```

Algorithm 19: VersionsToKeysGetKeys searches and retrieves the active keys in the provided versions range.

The algorithm traverses the AVL tree index and retrieves the bitmaps for the versions in the range [`verStart`, `verEnd`] (line 2), which are combined using a logical OR (line 3). The resulting bitmap is then traversed and for each set bit’s position, the key from the keys array at the same position is added to the `keys` set to be returned

(lines 4-6). These returned keys are traversed by the AMVSL index, where each key is queried in the provided versions range. Note that due to the immutability of the committed versions data, i.e., their corresponding bitmaps and the unique existing keys they represent, this data structure can be easily modified to support authentication and to persist on the blockchain if required.

We incorporate the VersionsToKeys data structure into our AMVSL index to enable efficient range search on versions range across all keys. Each call to the AMVSL upsert, delete, and commit operations is preceded with a call to the VersionsToKeys's corresponding operations. In the next sections all references to the AMVSL index refer to the AMVSL index incorporating the VersionsToKeys data structure.

8.2.9 Implementation of SVRK, MVRK, and MVAK queries

Algorithm 20: ProcessQuery

```

input : query, verStart, verEnd, keyStart, keyEnd, foundRows
1 if query = SVRK then
2   | // assuming verStart is the single version used
3   | rangeSearch(verStart, verStart, keyStart, keyEnd, foundRows);
4 else if query = MVRK then
5   | rangeSearch(verStart, verEnd, keyStart, keyEnd, foundRows);
6 else if query = MVAK then
7   | keys ← versionsToKeys.getKeys(verStart, verEnd);
8   | for key ∈ keys do
9   |   | rangeSearch(verStart, verEnd, key, key, foundRows);

```

The SVRK, MVRK, and MVAK queries are implemented using Algorithm 12: RangeSearch discussed in Section 14. Algorithm 20: ProcessQuery details their AMVSL implementation. `rangeSearch` requires a versions range $[verStart, verEnd]$ and a keys range $[keyStart, keyEnd]$. The SVRK query searches for a single version in a range of keys. To enable searching a single version (provided in the `verStart` parameter) it calls `rangeSearch` with the versions range $[verStart, verStart]$ and with the provided keys range parameters (lines 1-3). The MVRK queries a range of keys over a range of versions and is synonymous with Algorithm 12: RangeSearch (lines

4-5). The MVAK query (lines 6-9) only requires a versions range. It uses `Version-
sToKeysGetKeys` to retrieve all keys in the provided versions range (line 7). For
each retrieved key it calls `rangeSearch` with the provided versions range and the
keys range $[key, key]$ to search for the single key over the provided versions range
(lines 8-9).

8.3 Comparison to related work

In the related work (Section 3.3.1), Li et al. present an EMB-Tree, where for each
insert, update, and delete operation only the traversed path is used to recompute
the root hash as in AMVSL. Since the insert and delete operations in the EMB-
Tree are closely related to those in the B^+ -Tree, the operations may result in a
tree reorganization that modifies multiple nodes and traverse the resulting paths
to recompute the root hash. On the other hand, AMVSL uses a skip list-based
structure, where the insert and delete operations require the addition or deletion
of a single tower node (in the worst case scenario) with minimal recomputation
overhead. While the EMB-Tree supports point-queries as well as range queries it
does not support versioned data as does AMVSL.

In the related work (Section 3.3.2), we survey the SEBDB and FalconDB blockchain
databases that enable querying capabilities on top of a blockchain. These studies
are orthogonal to our present research. In this work, we do not build a full-fledged
blockchain-based database system, but rather provide AMVSL as a building block
to develop such a system with advanced query features.

8.4 Evaluation

8.4.1 Systems evaluated

We evaluate the proposed AMVSL approach against two existing approaches, namely MPT [137] and MBT [51]. Since these approaches only support point queries on latest data, for a fair comparison, we modified the MPT and MBT indexes to support multi-version range search capabilities, for which we describe the details next.

Unserializing child node addresses The AMVSL index does not use key/value serialization, thus we use MPT and MBT indexes implementation that do not use serialization to enable a faithful comparison. For the MPT index we modified the github implementation [101], where instead of storing a child branch serialized address, the direct reference is stored. For the MBT index we use an implementation provided by Seyyedemahsa Banihashemi and Bahman Jamshidi, which uses direct reference to child nodes.

Multi-version and range search support Both MPT and MBT indexes support point queries per key. However, since the indexes do not utilize the keys' ordering properties they cannot support key range queries. In addition, the two indexes do not support multi-versioning. To enable comparing these indexes with AMVSL, we modify them to support multi-versioning and range queries as we discuss next. We add an auxiliary tuples array index for versioned keys of the form $(key, version)$. When a versioned key is added to the MPT/MBT index it is added to the auxiliary index and its value is added to the MPT/MBT index at the scalar key $key||version$, where $||$ denotes concatenation. When the MPT/MBT index is queried for all values in the range of $([keyStart, KeyEnd], [versionStart, versionEnd])$ the auxiliary index is traversed. For each traversed $(key, version)$, if the following predicate is true: $(keyStart \leq key \leq KeyEnd) \wedge (versionStart \leq version \leq versionEnd)$,

Varied number of keys		Varied number of versions	
Subsets size (Unique values)	Batch size/ #versions	Subsets size (Unique values)	Batch size/ #versions
200K	10	100K	10
400K			100
600K			1000
800K			
1M			

Table 8.2: Synthetic uniform random dataset details

the MPT/MBT index is queried for the value of the scalar $key||version$. We use $key + ' | ' + version$ as the concatenation method.

8.4.2 Dataset

For evaluation purposes, we have generated a synthetic uniform random dataset composed of integer values in the range of 1 to 1 million. This dataset was further broken into several subsets (Table 8.2):

Varied number of keys - multiple subsets of varied sizes. Each subset is generated by extracting unique values using a uniform distribution without repetition. For each generated subset we create a batch, which contains 10 duplicates of the same subset.

Varied number of versions - a single subset of size 100K. The subset is generated by extracting unique values using a uniform distribution without repetition. From this subset we generate batches of varied sizes. Each batch contains a number of duplicates of the same subset.

8.4.3 Experimental setting

We compare the AMVSL and the modified MBT and MPT indexes performance on the SVRK, MVRK, and MVAK queries defined in Table 8.1. In the experiments, we use a row of two columns where the first column is a string key and the second column is an integer value. For all queries we measure the **average latency**, i.e., the time it takes a single query to execute, computed as the total time in milliseconds. For inserts, we measure the **average throughput**, i.e., how many operations are run

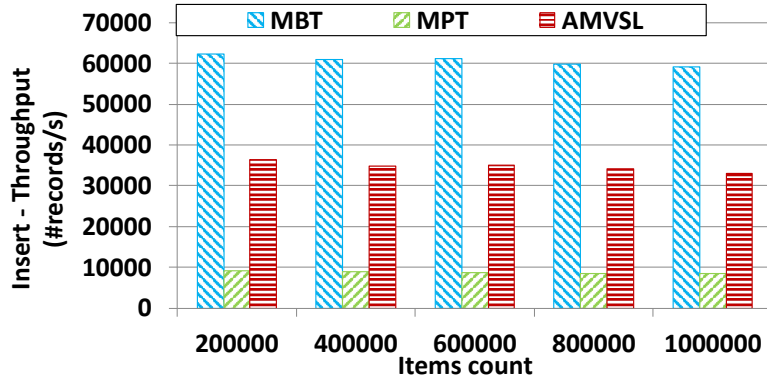


Figure 8.7: Insert throughput evaluation on “Varied number of keys” dataset

per second, computed as the number of executed operations/total time in seconds. Each evaluation is repeated 10 times and the average measurement is taken. Since the AMVSL index incorporates the VersionsToKeys index, which operates on strictly increasing consecutive keys, we sort the data on its keys before inserting it to the index. Each query that operates on a range of keys requires *keyStart* and *keyEnd* parameters. In each dataset setting, we provide this range by sorting the evaluated keys and extracting a subset of the first KP percent of the sorted keys. This subset’s first and last keys define the keys’ range, which is used as the *keyStart* and *keyEnd* parameters. A similar process is done to extract the subset of the first VP percent of the sorted versions to obtain the *verStart* and *verEnd* parameters from the first and last sorted subset items. For SVRK queries that focus on a single version across a range of keys, we set the first version to be the single version.

8.4.4 Experiments

8.4.4.1 Writes throughput on dataset “Varied number of keys”

In this setting the number of versions is fixed to 10 and the number of keys varies. Fig. 8.7 shows the insert throughput results of all indexes. The AMVSL index *consistently performs better* than the MPT index with an average performance increase of $4\times$. This may be due, in part, to the fact that MPT index’s write performance

is affected by the keys' length, unlike the AMVSL and MBT indexes. The MBT index performs better than the AMVSL index by an average of $1.75\times$. Part of this performance difference can be attributed to the additional two version fields in the AMVSL leaf nodes, which are cryptographically hashed in each write.

We note that all indexes' writes/reads duration, even of millions of records, are significantly overshadowed by the high duration overhead of the consensus protocol. Further, the MBT, MPT, and AMVSL indexes are designed for different scenarios. The MBT index is designed to be used online, and on current index data which benefits from higher write throughput, while the AMVSL index is designed to enable efficient online as well as offline key range queries across versions, where slightly slower writes can be tolerated in order to provide highly efficient range queries on current/historical authenticated data.

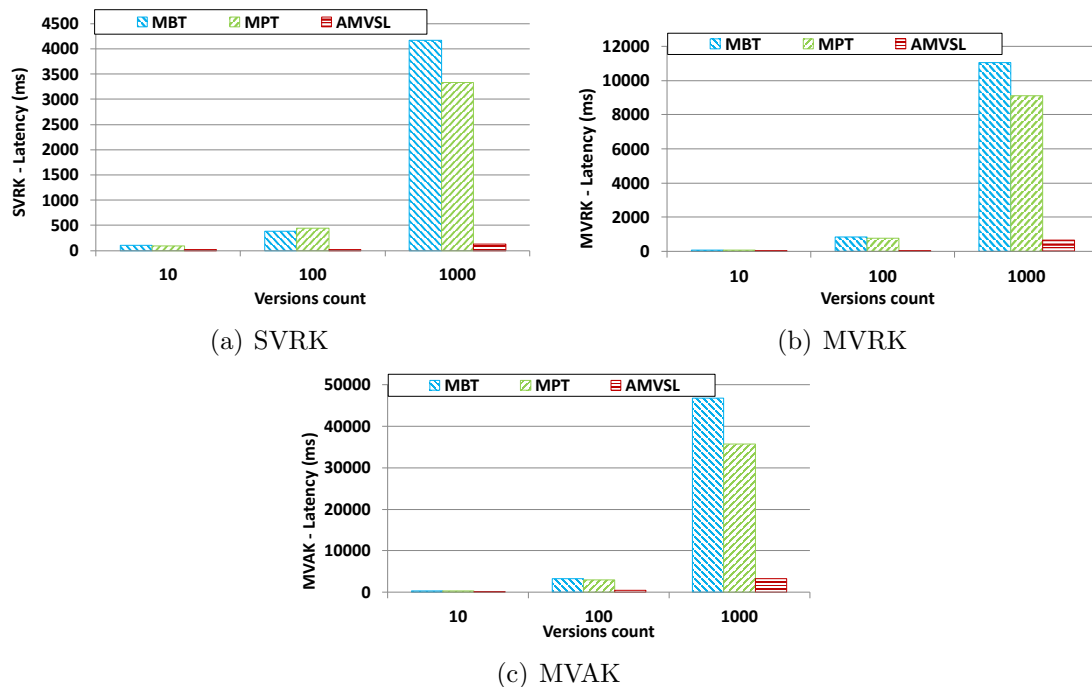


Figure 8.8: Query latency evaluation on dataset “Varied number of versions”

8.4.4.2 Query latency on dataset “Varied number of versions”

In this setting the number of keys is fixed to 100K and the number of versions varies. Fig. 8.8(a) shows the results for query SVRK, which queries the first version across 20% of the ordered keys. For versions 10 and 1000 using the AMVSL index is faster in comparison with the MPT index by $6\times$ and $25.6\times$ respectively. Fig. 8.8(b) shows the results for query MVRK, which queries 20% of the ordered keys across 20% of the ordered versions. For versions 10 and 1000 using the AMVSL index is faster in comparison with the MPT index by $6.4\times$ and $13.8\times$ respectively. Fig. 8.8(c) shows the results for query MVAK, which queries 20% of the ordered versions across all keys. For versions 10 and 1000 using the AMVSL index is faster in comparison with the MPT index by $2.2\times$ and $10.9\times$ respectively. The reason for the significant increase in performance is due to the fact that the increase in versions affects only the versions search in AMVSL and not the keys search. On the other hand, the increase in versions increases the total versioned keys processed by the MBT and MPT indexes (by an order of magnitude between sequential versions count), which significantly affects their query performance.

8.4.4.3 Query latency on dataset “Varied number of keys”

In this setting the number of versions is fixed to 10 and the number of keys varies. Fig. 8.9(a) shows the results for query SVRK. For rows count 200K and 1M using the AMVSL index is faster in comparison with the MPT index by $4.8\times$ and $6.2\times$ respectively. Fig. 8.9(b) shows the results for query MVRK. For rows count 200K and 1M using the AMVSL index is faster in comparison with the MPT index by $5.2\times$ and $5.8\times$ respectively. Fig. 8.9(c) shows the results for query MVAK. For rows count 200K and 1M using the AMVSL index is faster in comparison with the MPT index by $1.9\times$ and $2.2\times$ respectively.

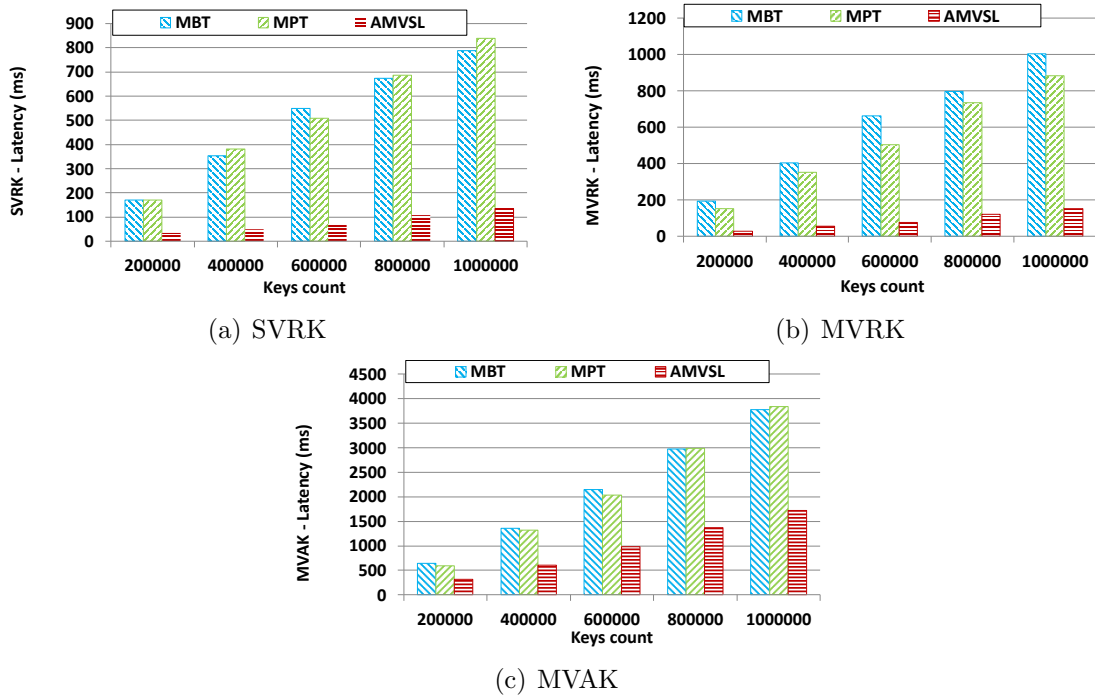


Figure 8.9: Query latency evaluation on dataset “Varied number of keys”

8.4.4.4 Query latency on 1M keys, 10 versions, and varied keys/versions search percentage

In this setting we evaluate the queries on a subset of the dataset “Varied number of keys” where the number of versions is fixed to 10, the number of keys is fixed to 1M, and we vary the percentage of searched keys/versions. Fig. 8.10(a) shows the results for query SVRK. For keys percentage 0.1 and 0.5 using the AMVSL index is faster in comparison with the MPT index by $10\times$ and $4.7\times$ respectively. Fig. 8.10(b) shows the results for query MVRK. For keys percentage 0.1 and 0.5 using the AMVSL index is faster in comparison with the MPT index by $6.8\times$ and $8.3\times$ respectively. Fig. 8.10(c) shows the results for query MVAK. For keys percentage 0.1 and 0.5 using the AMVSL index is faster in comparison with the MPT index by $1.3\times$ and $4.4\times$ respectively.

In queries SVRK and MVRK, the AMVSL index significantly outperforms the MBT and MPT indexes, specifically, when the keys and versions count increase. This is

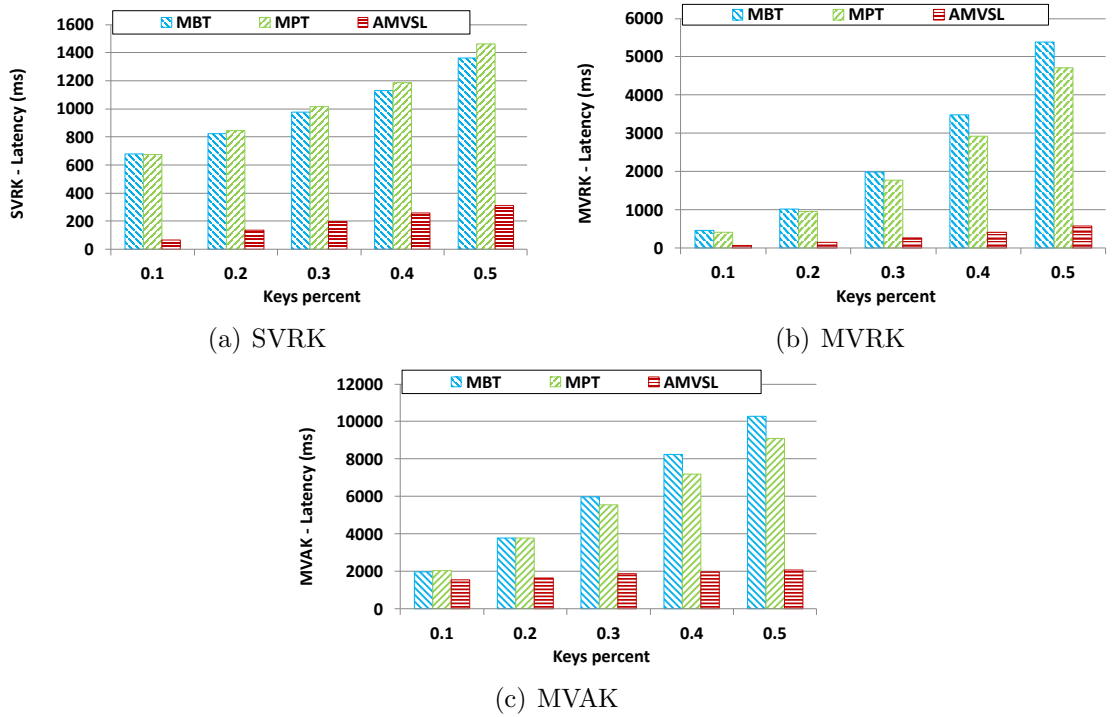


Figure 8.10: Query latency evaluation on 1M keys, 10 versions, and varied keys/versions search percentage

due to how each versioned key value is first located and then retrieved by each index. In the versioned MPT/MBT indexes the versioned keys' auxiliary data structure is traversed. Each traversed versioned key that is in the provided keys/versions range is queried by the index to retrieve its corresponding value. This results in traversing MPT/MBT index nodes multiple times for all queried keys. Conversely, in the AMVSL index, all nodes are traversed exactly once. The performance difference between the indexes in the MVAK query is slightly lower compared to the SVRK and MVRK queries. This is due to having the AMVSL index using the versionsToKeys data structure to locate the keys across the provided versions range, which results in a discrete key set. For each key in this set a point query is performed in the AMVSL index where some nodes may be traversed more than once.

8.4.5 Complexity Analysis

We conducted a systematic analysis of the upsert and query algorithms of AMVSL and our modified MBT and MPT indexes that support versioning and range queries.

8.4.5.1 AMVSL

We start by analyzing the complexity of each data structure and continue to describe the complexity of the whole comprising indexes.

AMVSL skip list The key search cost in the AMVSL's skip list is the same as that of a random skip list search, which is $O(\log K)$, where K is the number of keys in the index. The upsert operation includes traversing the path to the desired row's location with a cost of $O(\log K)$. At the worst case a new tower is inserted with a cost of $O(1)$. The authentication computation of each traversed tower level when backtracking the same path costs an additional $O(\log K)$. Hence, the overall upsert cost is $O(\log K + \log K) = O(\log K)$.

AMVSL bucket Due to the bucket's row entries being inherently sorted on the `validFrom` field, a single version can be searched using a binary search with no additional sorting, with a cost of $O(\log V)$, where V is the number of all versions. When a range of versions are searched, the first range is searched with a cost of $O(\log V)$, from which all higher versions are sequentially searched until the current version is bigger than the range end key or all versions were searched with an overall cost of $O(\log V + V_r)$, where V_r is the number of versions in the versions range.

The upsert operation adds a new row entry to the top of the bucket or updates the topmost row entry. The digest of the top row entry is dependent only on the digest of the previous row entry with a total cost of $O(1)$.

AMVSL index The AMVSL index's range search over a range of keys and versions is composed of first locating the start range key, with a cost of $O(\log K)$ and traversing the index from that key over the sequential keys until the currently traversed key is bigger than the end range key with an overall cost of $O(\log K + K_r)$ where K_r is the number of keys in the keys range. For each range key, the corresponding bucket is searched for the range of versions as described above. Hence, the overall search cost for a range of keys and versions is $O(\log K + K_r(\log V + V_r))$. The upsert operation is composed of traversing the AMVSL skip list and upserting the located tower bucket's topmost row with an overall cost of $O(\log K + 1) = O(\log K)$.

8.4.5.2 MBT and MPT (without multi-version or range search support)

The MBT and MPT share similar operations complexity. The search cost is $O(\log K)$, where K is the number of keys in the index. No versioning or range queries are supported by MBT and MPT. The upsert operation includes traversing the path to the desired key's location with a cost of $O(\log K)$. The operation itself is performed with a cost of $O(1)$. The authentication computation of each traversed node when backtracking the same path costs an additional $O(\log K)$. Hence, the overall upsert cost is $O(\log K + \log K) = O(\log K)$.

8.4.5.3 MBT and MPT with multi-version and range search support

The MBT and MPT with multi-version and range search support share similar operation complexity. In order to search for a versioned key, the auxiliary versioned keys array is traversed with a cost of $O(K_v)$ where, K_v is the number of versioned keys. Next, the versioned key is searched in the index with a cost of $O(\log K_v)$ with an overall cost of $O(K_v + \log K_v)$. Range search over a range of keys and versions is composed of traversing the auxiliary versioned keys array. If a traversed item's versioned key is in the keys and versions range its value is searched for in the index

Index	Upsert	Search	Range search
MBT	$O(\log K)$	$O(\log K)$	-
MBT supporting MV-range search	$O(\log K_v)$	$O(K_v + \log K_v)$	$O(K_v \log K_v)$
MPT	$O(\log K)$	$O(\log K)$	-
MPT supporting MV-range search	$O(\log K_v)$	$O(K_v + \log K_v)$	$O(K_v \log K_v)$
AMVSL	$O(\log K)$	$O(\log K)$	$O(\log K + K_r(\log V + V_r))$

Table 8.3: Complexity analysis summary

Parameter	Description
K	Number of keys
V	Number of versions
K_v	Number of versioned keys
K_r	Number of keys in the keys range
V_r	Number of versions in the versions range

Table 8.4: Complexity parameters description

with an overall cost of $O(K_v \log K_v)$. When inserting a versioned key, the tuple (key, version) is added to the auxiliary versioned keys array with a cost of $O(1)$. The upsert operation includes traversing the path to the desired versioned key's location with a cost of $O(\log K_v)$. The operation itself is performed with a cost of $O(1)$. The authentication computation of each traversed node occurs when backtracking the same path with an additional cost of $O(\log K_v)$. Hence, the overall upsert cost is $O(\log K_v + \log K_v) = O(\log K_v)$. Table 8.3, and its legend in Table 8.4, shows the complexity summary of the AMVSL, MPT, MBT indexes, and our modified MBT and MPT indexes that support versioning and range queries, on their different operations.

Chapter 9

Scalable Privacy-Preserving Query Processing Over Ethereum Blockchain

Private blockchains contain sensitive information, which may need to be audited according to regulatory requirements. The auditors need to have access to a company's raw blockchain data, as opposed to an intermediate processed data repository, where the data may have been tampered with. In addition, to support more efficient and rigorous audit procedures, the disclosure of queries should be prevented. As the amount of data stored in the blockchain increases, the immutability of every block and its transactions ensures a rapid data size increase. Due to the linked structure of the blockchain, only sequential pass over the entire blockchain data is possible, which limits querying capabilities. In order to enable richer and more performant querying, auditors need to fetch the raw data and then process it offline. This can be achieved by implementing a capable server with access to a company's blockchain nodes.

As an Ethereum node can be implemented in various languages, according to the

Ethereum yellow paper [137], we chose to focus on Go-Ethereum (Geth), which stores data in a key-value store (LevelDB [90]). The Ethereum node uses a general JSON RPC protocol [61], which specifies limited querying capabilities of its internal storage. These support the retrieval of one block or transaction per request. In order to retrieve multiple blocks or transactions, multiple API calls per block/transaction need to be executed. This can be inefficient, especially as the blockchain constantly expands in data volume as new blocks are added to the chain. Several third party tools have been developed to address the scalability issues in the form of a centralized service. For example EtherQL [93] downloads the Ethereum blockchain data, stores it in MongoDB and exposes an API with predefined queries. However, custom queries and private information retrieval are not supported. Another system, vChain [138] provides a way to execute boolean range queries using cryptographic proofs, vital to enable query integrity. However, the use of cryptographic proofs incurs high processing times, which are orders of magnitude slower compared to our proposed system. Similar additional tools are described in the related work Section 3.3.3.

In this chapter, we propose a system that enables multiple auditors to perform richer queries over blockchain data in an efficient and scalable way. Our system additionally supports private information retrieval by utilizing cryptography techniques over semi-trusted servers to protect the auditors' identities, queries, and their results. To handle the current and rapidly increasing blockchain data volume, the system employs Hadoop [12], which is a scalable distributed processing solution for big data. Users submit SQL queries which are transformed into MapReduce tasks and are run on Hadoop. When missing data are required MapReduce tasks are created and used to download the data from Ethereum nodes in parallel and store them in the local Hadoop Distributed File System (HDFS). An in-memory B^+ Tree-based index is used to index the downloaded data for efficient future access. The entire data fetching process utilizes privacy-preserving techniques. The client and the data server share

a secret key. The client sends the server an encrypted and modified query, the server sends the client the encrypted results and the client continues to further refine the results according to the full SQL query. The communication between the client and the server involves an intermediary proxy, which also serves to hide the identity of the client. Our contributions can be summarized as follows:

1. Enable support for SQL query language over blockchain data, which includes SELECT statements and aggregate functions, e.g., MIN, MAX and SUM, with WHERE clauses to fetch blockchain blocks and transactional information over specified ranges and additional filters.
2. Design and implement a scalable and robust system that executes queries in a parallel and distributed fashion by utilizing Hadoop's MapReduce infrastructure.
3. Design and implement a private information retrieval approach to ensure the client's (auditor) confidentiality in the submitted query and its results during the communication between the different parties, i.e., client, proxy and the data server.

9.1 The proposed system

To better illustrate our approach we use a motivating scenario, for which the sequence diagram is provided in Fig. 9.1. In the sequence, the user (auditor) sends the SQL query “SELECT MAX(value) FROM transactions WHERE block_number BETWEEN 100 AND 200” to the Client. The Client parses the query, extracts a partial query with an extended range “SELECT * FROM transactions WHERE block_number BETWEEN 87 and 2061”, and sends it to the Proxy in a privacy preserving way, which sends it to the Server. The Server efficiently retrieves the data required by the extended range query and sends it to the Proxy, in a privacy preserving way, which sends it to the Client. The Client runs the remaining parts of the original SQL query

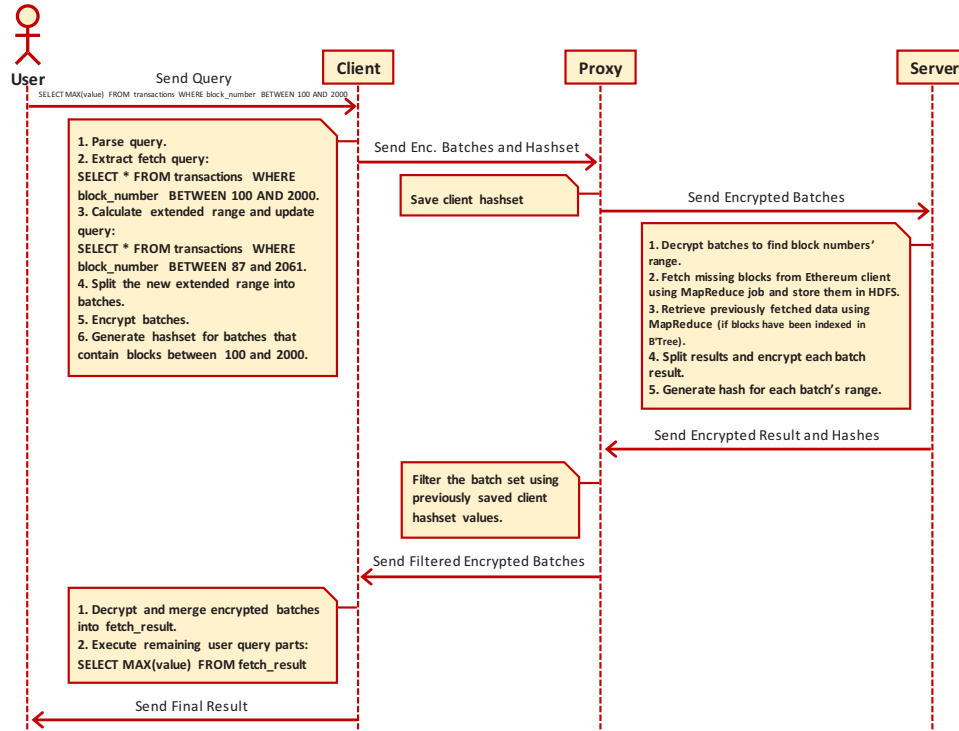


Figure 9.1: Flow of user query execution: `SELECT MAX(value) FROM transactions WHERE block_number BETWEEN 100 AND 2000`

to get the final results as shown in the following query: “`SELECT MAX(value) FROM fetched_results`”.

9.1.1 Main components

Client - Parses the user’s (auditor) query, which includes a *block_number* range, and extracts a *fetch query* and a *processing query*. The *fetch query* is used to fetch all blocks/transactions required by the *processing query* while enforcing privacy preservation. The *processing query*, which includes the main query logic is run on the fetched data.

Proxy - Acts as a mediator between the client and server. It hides the source of the client from the server and filters the fetched data from the server before sending it to the client in order to save network bandwidth and decryption resources.

Data Processing Server - Serves blocks or transactions requested by the client’s

modified *fetch query*. The data retrieval includes fetching missing data, if required, from Geth clients and saving them for future use. These are done using Hadoop MapReduce tasks. The results are encrypted and sent securely to the proxy.

9.1.2 Assumptions:

- The client and server share a secret key for data encryption/decryption.
- The proxy and the server are assumed to be honest-but-curious; i.e., the two follow the protocol, but the proxy may be curious about the content of the retrieved confidential results from the server, while the server may be interested in the exact range of the queries sent by the client. If an organization on the proxy side can gain access to the exact results it can extract confidential information about the company data that were extracted by the server. If an organization on the server side can gain information to the exact queried range it can deduce more accurately what specific blocks or transactions are of interest to the auditors, which can help it to more efficiently prepare a response if the queried data may contain information of illicit activities. By splitting the query.
- No collusion between the proxy and the server. Since the server decodes the *fetch query*, it knows only the extended range of the blocks/transactions from the client's query. The proxy contains knowledge of the actual range in the user's query. Collusion between the server and the proxy can reveal identities of the auditors and expose the exact queried range of results. Therefore, it is crucial that the server and the proxy do not share information to prevent any potential breaches of confidentiality.

9.1.3 Privacy preserving query processing

To preserve the privacy of the user's query and to securely receive the results, the client splits, modifies and encrypts the user's query. This is done by extracting required block range only from the query, obfuscating this range with extended lower and upper dummy boundaries, and encrypting the extended block range by applying a secret key that is shared between both client and server. The client sends this partial, modified, and encrypted query to the proxy.

The proxy cannot decrypt the query and can only propagate the encrypted query to the server. The server decrypts the partial query by using the shared key. The decoded partial query seen by the server includes only the extended range of blocks, from which it cannot deduce the user's original full query including its exact range. In addition, since the proxy serves as an intermediary between the client and the server, the server cannot identify the query sender (user).

The server then retrieves the blocks and transactions according to the received query's range, encrypts them, and sends back the encrypted result to the proxy. The proxy cannot decrypt the results but can filter some unnecessary results to improve the communication and decryption performance. The proxy then sends the filtered encrypted results to the client, which decrypts the results on which it continues to execute the complete query.

9.2 System Model

The overall system architecture is shown in Fig. 9.2. In this section we describe the design and implementation of the proposed system and explore in detail the processes involving all of its components.

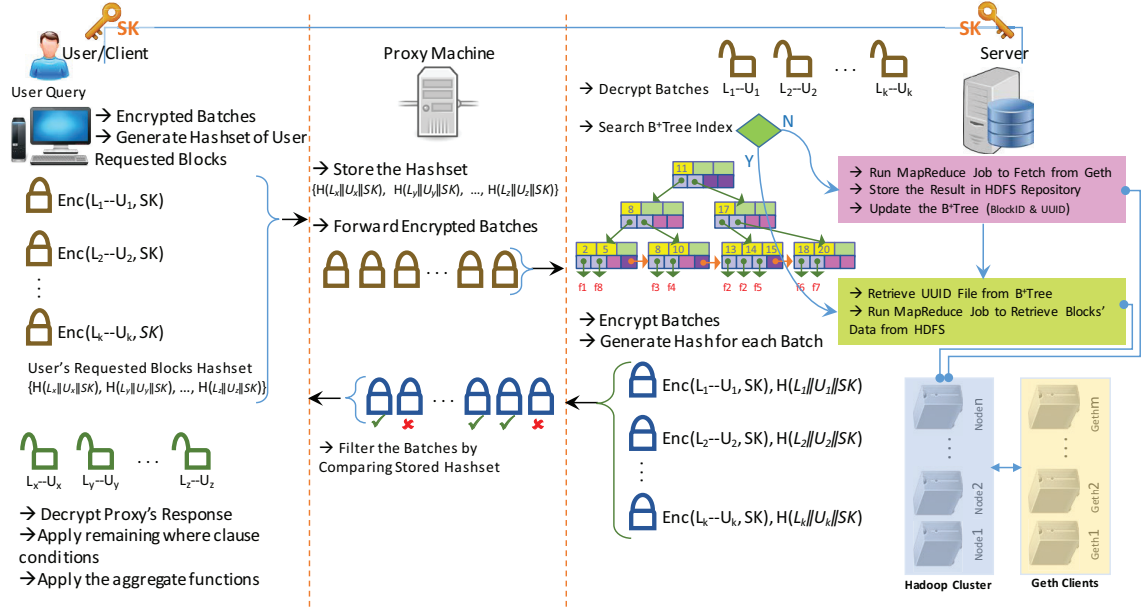


Figure 9.2: System Components and Model

Q1	SELECT MAX(value) FROM transactions WHERE block_number BETWEEN <LB> AND <UB> AND account_address=<address>
Q2	SELECT * FROM transactions WHERE block_number BETWEEN <LB> AND <UB> AND account_address=<address>

Table 9.1: Query examples

SQL query parser The client parses the user’s query using a parser built with ANTLR4 [32] and supports SELECT statements that include aggregate functions, e.g., MIN, MAX, and SUM, and WHERE clauses including range specifications. The FROM clause can receive either blocks or transactions as a source. We demonstrate our system’s capabilities with two example queries (inspired by [53]) as presented in Table 9.1. The motivating scenario sequence uses the Q1 query form (Table 9.1).

Client to proxy The client receives the query from the user, parses it using the SQL query parser and extracts two queries: a *fetch query* and a *processing query*.

FQ	SELECT * FROM transactions WHERE block_number BETWEEN <ELB>AND <EUB>
PQ	SELECT MAX(value) FROM fetched_results WHERE account_address=<address>

Table 9.2: Query Q1 Fetch/Processing extracted queries example

The *fetch query* is used to fetch all blocks in the extended range from the server. Once the fetched data are received by the client, the *processing query* processes the data locally. For example, the query Q1 in Table 9.1 is split into a *fetch query* (FQ in Table 9.2) whose results are stored locally in *fetched_results* and to a *processing query* (PQ in Table 9.2). To generate the *fetch query* (Algorithm 21), the client

Algorithm 21: client_prepare_fetch_query

input : SQL_like_query - user query, Sk - client/server shared key, LBR/UBR - Lower/Upper Bound Range to compute random value, blocks_per_batch - number of blocks per range split
output: enc_range_splits_list - list of encrypted ranges, ranges_hash_set - hashes of ranges intersecting query range

```

1 fetch_query, process_query ← parse(SQL_like_query);
2 LB, UB ← parse(fetch_query);
3 rand_lower ← get_random_in_range(LBR);
4 rand_upper ← get_random_in_range(UBR);
5 ELB ← min(1, LB - rand_lower);
6 EUB ← UB + rand_upper;
7 ranges_hash_set ← {};
8 enc_range_splits_list ← list();
9 extended_range_size ← (EUB - ELB + 1);
10 splits_count ← extended_range_size/blocks_per_batch;
11 for i ← 0 to splits_count by 1 do
12     split_start ← ELB + i × blocks_per_batch;
13     split_end ← split_start + blocks_per_batch;
14     enc_range_splits_list, ranges_hash_set ← get_range_split(LB, UB, split_start, split_end, Sk);
15 // last remaining range split is addressed similarly

```

extracts the *block_number*'s lower/upper bounds range ($[LB, UB]$) from the query. The $[LB, UB]$ range is then extended by subtracting a user defined random value from LB and adding a different user defined random value to UB , resulting in the extended $[ELB, EUB]$ range. This extended range is then split into a set of ranges with a user defined batch size, which is encrypted individually using a secret key Sk that is shared with the server. In addition, for each block range that intersects

Algorithm 22: get_range_split

input : LB/UB - query Lower/Upper Bound, split_start - split lower bound block number, split_end - split upper bound block number, Sk - client/server shared key
output: enc_range_splits_list - encrypted ranges, ranges_hash_set - contains only hashes of ranges intersecting the query's original range

```

1 range ← (split_start, split_end);
2 enc_range ← AES.encrypt(range, Sk);
3 enc_range_splits_list.add(enc_range);
4 if (split_start ≤ LB ∧ split_end ≥ LB) ∨ (split_start ≥ LB ∧ split_end ≤ UB) ∨ (split_start ≤ UB ∧ split_end ≥ UB) then
5     range_hash ← SHA256(range|Sk);
6     ranges_hash_set.add(range_hash);

```

the range of the the original $[LB, UB]$ range, which is appended with Sk , the client generates a hash (Algorithm 22). The client then sends the encrypted split range batches and their hashes to the proxy. The proxy receives the encrypted range splits and their hashes and saves the hashes for future processing.

Index The server uses an in-memory B^+ Tree-based index, which is indexed on block number to retrieve the corresponding transactions' data file paths in HDFS. The index serves two purposes when used on a range of block numbers:

1. It finds the HDFS repository file paths that contain the transaction/block in the provided range.
2. It finds block numbers, which do not exist in the HDFS repository in the provided range.

Proxy to server The proxy propagates the encrypted range splits to the server. The server decrypts the encrypted query, which consists of the block number extended ranges, for which it needs to retrieve the corresponding data. The server uses HDFS to store blocks/transactions and to run queries using MapReduce tasks in order to retrieve existing data from the HDFS repository and to fetch missing data from Geth clients.

To serve queries that contain a block number range, the server executes two MapReduce tasks (Algorithm 25):

1. A MapReduce task to fetch missing blocks/transactions that are not already stored in the HDFS repository from Geth [6] clients using Web3j [136] (Algorithm 23).
2. A MapReduce task to retrieve existing blocks/transactions from the HDFS repository (Algorithm 24).

Algorithm 23: server_fetch_data_from_Geth

```
input : missing_blocks_list - missing blocks to fetch from Geth clients, blocks_per_task - blocks to fetch
        per task, Geth_client_ip_addresses - ip addresses of Geth clients, threads_per_task - number of
        threads per task
1  tasks_batches ← list();
2  batch ← list();
3  batch_count ← missing_blocks_list.size()/blocks_per_task;
4  for i ← 0 to batch_count by 1 do
5    batch_start ← i × blocks_per_task;
6    batch_end ← batch_start + blocks_per_task;
7    for j ← batch_start to batch_end by 1 do
8      block_number ← missing_blocks_list.get(j);
9      batch.add(block_number);
10   tasks_batches.add(batch);
11   batch ← list();
12  if missing_blocks_list.size()/blocks_per_task ≠ 0 then
13    batch_start ← batch_count × blocks_per_task;
14    batch_end ← missing_blocks_list.size();
15    for j ← batch_start to batch_end by 1 do
16      block_number ← missing_blocks_list.get(j);
17      batch.add(block_number);
18    tasks_batches.add(batch)
19  node_Geth_client_config_list ← list();
20  i ← 0;
21  for task_batch in tasks_batches do
22    i ← (i + 1)%Geth_client_ip_addresses.size();
23    Geth_client_ip_address ← Geth_client_ip_addresses.get(i);
24    node_Geth_client_config ← (Geth_client_ip_address, threads_per_task, task_batch);
25    node_Geth_client_config_list.add(node_Geth_client_config);
26  // copy node_Geth_client_config_list to HDFS file: node_Geth_client_config_list_file
27  fetch_from_Geth_clients_map_reduce.task(node_Geth_client_config_list_file, HDFS_output_dir);
28  // copy all mappers results files from HDFS_output_dir to local_results_dir
29  merged_file ← generate UUID;
30  // merge the files in local_results_dir to merged_file
31  // copy merged_file to HDFS with the same file name
32  Index.update(missing_block_numbers, merged_file);
33  // (an offline process partitions merged_file per block number and re-arranges the index)
```

Upon receiving the encrypted query from the proxy containing the extended block numbers' range splits (Algorithm 25), the server decrypts the ranges and extracts the extended lower/upper bounds of the extended range $[ELB, EUB]$. It then searches the index for the block numbers that are missing in the index, and consequently, in HDFS. If there are such blocks (Algorithm 23), a MapReduce task is run to fetch the missing blocks/transactions from the Geth clients. To do so the server prepares an input file for the MapReduce task. The MapReduce task is configured to consume the input file one line at a time. Each input file line contains a different iterating Geth client IP address in order to distribute the fetching load between all tasks, in addition to a user defined *threads_per_task* count to use in each node to fetch

Algorithm 24: server.retrieve_data_from_Hadoop

```
input : ELB/EUB - Extended Lower/Upper Bound, range_splits_list - range splits, HDFS_output_dir,
        local_results_dir, block_number_pos - position in results line
output: enc_ranged_res_hash_map - all blocks in range, split by ranges, encrypted and hashed
1 index_file_name_list  $\leftarrow$  Index.get_indexed_file_names(ELB, EUB);
2 update retrieve_map_reduce template source code to fetch only data in the block numbers range
  [ELB, EUB];
3 compile the updated map_reduce code into updated_retrieve_map_reduce;
4 updated_retrieve_map_reduce.task(index_file_name_list, HDFS_output_dir);
5 copy all mappers results files from HDFS_output_dir to local_results_dir;
6 results  $\leftarrow$  merge all mappers results files in local_results_dir;
7 ranged_results_hash_map  $\leftarrow$  {};
8 for range in range_splits_list do
9   | ranged_results_hash_map.add(range, list());
10 sorted_range_splits_list  $\leftarrow$  sort range_splits_list on LB;
11 for res in results do
12   | block_number  $\leftarrow$  res.get(block_number_pos);
13   | range  $\leftarrow$  sorted_range_splits_list.get_range(block_number);
14   | //get_range function searches sorted_range_splits_list for a range with the largest LB that is
15   |   smaller or equal to block number. Works in  $\mathcal{O}(\log n)$ 
16   | ranged_results_hash_map.get(range).add(res);
17 enc_ranged_res_hash_map  $\leftarrow$  {};
18 for range in ranged_results_hash_map.keys do
19   | range_hash  $\leftarrow$  SHA256(range|Sk);
20   | batch  $\leftarrow$  ranged_results_hash_map.get(range);
21   | enc_batch  $\leftarrow$  AES.encrypt(batch, Sk);
22   | enc_ranged_res_hash_map.add(range_hash, enc_batch);
```

the blocks concurrently. The number of input file lines and, hence, the number of tasks is computed by $missing_blocks_list.size()/blocks_per_task$. The *blocks_per_task* parameter should be determined in advance to maximize memory consumption in each node in order to utilize the nodes efficiently. The MapReduce task is configured to not use reducers in order to prevent memory issues when downloading a large amount of data and to maximize resources usage (more nodes are used as mappers) when fetching the data from Geth clients. Each mapper communicates with a synchronized Geth client using Web3j to fetch the missing data and produces a result file in the *HDFS_output_dir* directory. When all MapReduce tasks are complete, the mappers' result files are downloaded locally to the server from HDFS and merged into one file that contains all blocks/transactions delimited by a new line. The file is then uploaded to the HDFS repository to be used in future queries. The index is updated to point to the uploaded file, which contains the missing blocks. An offline process is used to split this file into smaller files and update the index accordingly

Algorithm 25: *server_get_data*

input : *enc_range_splits_list*, *Sk* - client/server shared key, *blocks_per_task*, *Geth_client_ip_addresses*,
threads_per_task, *HDFS_output_dir*, *local_results_dir*, *block_number_pos* - position in results line
output: *enc_ranged_res_hash_map* - all blocks in range, split by ranges, encrypted and hashed

```
1 range_splits_list ← list();
2 for enc_range in enc_range_splits_list do
3   | range ← AES.decrypt(enc_range, Sk);
4   | range_splits_list.add(range);
5 ELB ← range_splits_list.first().LB;
6 EUB ← range_splits_list.last().UB;
7 missing_blocks_list ← Index.get_missing_blocks(ELB, EUB);
8 if missing_blocks_list ≠ ∅ then
9   | server_fetch_data_from_Geth(missing_blocks_list,  
   | blocks_per_task, Geth_client_ip_addresses, threads_per_task);
10 enc_ranged_res_hash_map ← server_retrieve_data_from_Hadoop(ELB, EUB,  
   range_splits_list, HDFS_output_dir, local_results_dir, block_number_pos);
```

for more efficient retrievals in future queries.

In the next step, the server retrieves the blocks requested by the query from the HDFS repository, encrypts them, generates their hashes, and returns both to the proxy. To do so (Algorithm 24), the server searches the index for the HDFS repository indexed file paths that contain all block numbers in the extended query range. The resulting list of file paths is used as input to the MapReduce task that retrieves blocks/transactions from HDFS. To prepare this MapReduce task, the server uses a Java template code snippet that provides the functionality for a task to get each line from its input file and output the line only if it passes a filter statement. The filter statement is a placeholder for an if-condition that is generated by the server according to the extended lower and upper bounds extended range $[ELB, EUB]$ of the query. The code is then compiled to produce the mapper task binaries. This MapReduce task is also configured to not use reducers to preserve memory and maximize node resources. Each mapper uses a text file input split with a default split size of 64M. The task is then run with the indexed file paths retrieved from the indexer and the mappers' results files are copied to the server locally and merged into a single file whose lines are delimited by a new line. The merged results file is then partitioned into batches. Each batch contains block lines with block numbers in the batch's specific range. The ranges are extracted from the range splits provided

in the query. Each batch is encrypted using the secret key shared with the client and the hash of the batch range plus the secret key is computed.

Algorithm 26: proxy_filter_results_from_server

input : *enc_ranged_res_list* - contains tuples of (*split_range_hash*, *range_split_enc_res*) where *range_split_enc_res* is the encrypted results corresponding to a specific range split that is hashed in *split_range_hash*. The ranged splits are in the same structure as provided by the client, *ranges_hash_set* - contains the hashes of the client's desired range splits

output: *filtered_range_split_enc_res* - contains only the encrypted split range results according to the provided split range hash that was provided by the client in *ranges_hash_set*

```

1 filtered_range_split_enc_res ← list();
2 for results in enc_ranged_res_list do
3   | if results.split_range_hash in ranges_hash_set then
4   |   | filtered_range_split_enc_res.add(results.range_split_enc_res);

```

Server to proxy The server sends the encrypted results and their hashes to the proxy. The results are composed of tuples. Each tuple contains an encoded batch with block/transaction lines, and the second item in the tuple is the hash of the batch range plus the shared key. The proxy filters the received results by retaining only the batches for which the range hash is contained in the hashes provided by the client (Algorithm 26). This saves network bandwidth when the proxy sends only relevant results to the client and saves decryption time when the client decrypts the received results. The size of a batch (containing a specific range) affects the overall encryption/decryption performance. The encryption/decryption time grows significantly with larger batches. Subsequently, smaller batches are used.

Proxy to client The proxy sends the filtered results to the client. The client decrypts each batch with the shared key and merges it into a *fetch_results* list. Since a batch may contain information on block numbers not in the user's full query range, the client filters these lines. The client continues to locally execute the process query on the *fetch_results*, which includes the aggregate functions and other WHERE clause filters.

Blocks	LB	UB	Blocks	LB	UB
100	3000000	3000100	200K	3000000	3200000
1K	3000000	3001000	300K	3000000	3300000
10K	3000000	3010000	400K	3000000	3400000
100K	3000000	3100000	500K	3000000	3500000

Table 9.3: Block numbers and ranges used

9.3 Comparison to related work

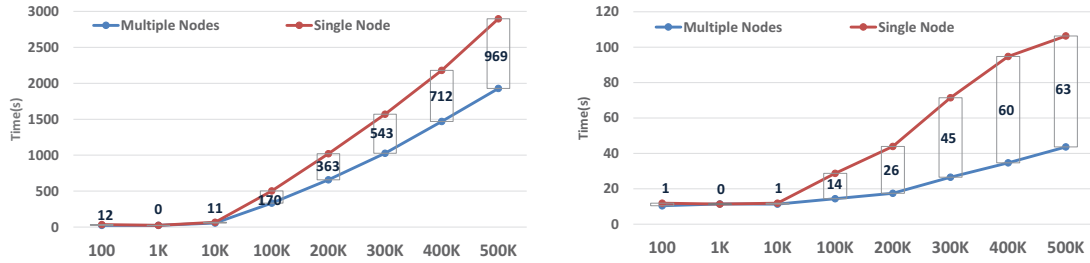
We compare our proposed approach to the surveyed approaches in the related work (Section 3.3.3). Most of the approaches are designed to query public blockchains. The use of a limited API on a single node contributes to limited querying capabilities that lack retrieval efficiency using multiple logical combinations. On the other hand, VChain [138] enables privacy preserving boolean query results, which requires modification of the full node. Our solution is designed to be used “on top” of blockchain systems without requiring their modification. Whereas vChain requires a query to be in a limited format, our solution supports more versatile SQL queries. Also, vChain uses homomorphic encryption techniques, which are costly compared to our solution’s use of lightweight symmetric encryption AES and SHA256. In addition, vChain performs all cryptographic computations on the server containing the full node; in our solution the *fetch query* execution initiates the encryption and download of the ranged data from a blockchain node and eventually continues with the processing of the query’s main logic on the client side. This results in processing times that are split between the client and server that are significantly lower compared to vChain. Finally, the SQL query in our framework provides more capabilities (e.g., aggregation), which can be easily extended to support more complex features that are not restricted by the homomorphic encryption constraints.

9.4 Evaluation

In this section we evaluate our system in various settings. We first describe the experimental setup, datasets, and configuration parameters used in our experiments.

Nodes	Geth clients	Blocks per batch	Threads per task
4	2	5000	4
1	1	5000	8

Table 9.4: Single and multiple nodes configuration



(a) Fetching from Ethereum Geth clients on system initialization

(b) Fetching from (already indexed) HDFS

Figure 9.3: Evaluation of fetched blocks data from (a) Geth clients and (b) HDFS

The experiments were conducted on a cluster of four machines, each having an Intel(R) Xeon(R) CPU E5472 @ 3.00GHz and 8GB of memory. Two additional machines were used for running Ethereum Geth clients. We used a dataset from live Ethereum feed provided by the Geth clients. Table 9.1 shows the queries used for evaluation. Table 9.3 shows the varied block numbers and their ranges values used in our experiments. Each block range configuration was run with either 1 or 4 nodes per cluster (Table 9.4). “Blocks per batch” indicates the number of blocks each task downloads. “Threads per task” indicates the number of threads each task uses to download the transactions/blocks concurrently from the Geth clients.

The experiments evaluate two independent processes:

1. Server fetch time, which consists of fetching missing blocks, index update, storing of newly fetched data into HDFS, and fetching all query data from HDFS.
2. All steps of the query processing time after the server’s data fetch. This includes server results encryption, proxy filtering, client results decryption, applying WHERE clause filtering, and aggregate function.

The server’s total fetch time is dependent only on the query’s extended range size and not on any other part of the query. This means that different queries with the same

extended range should present similar server total fetch times. Fig. 9.3 demonstrates the significant improvement of our solution due to the parallel downloading and fetching of the query blocks/transactions.

9.4.1 Fetching missing data from Geth clients

When the system is initialized (first time), all Ethereum blocks generated until that point are needed to be fetched from the Geth client(s) and the index needs to be built. After that, a background process is run periodically to fetch newer blocks from Geth clients. Fig. 9.3(a) shows the time improvement when all blocks are fetched from the Geth client(s) by using multiple Hadoop nodes in comparison to that with a single node. The average speedup achieved between the single and multiple nodes configurations is $1.52\times$.

In this configuration, 2 Geth clients are used with the 4 Hadoop nodes. It may seem that the speedup should be at least 2. However, since all 4 nodes send requests to the same 2 Geth clients, the consequent load on each Geth client lowers the speedup gain. Using at least 2 more Geth clients would significantly increase the speedup.

9.4.2 Fetching data from HDFS repository

For most queries, it is expected that the blocks specified by the query ranges are already fetched from Geth clients. Fig. 9.3(b) shows the time improvement for the data retrieval from HDFS, when all blocks are already fetched from the Geth clients and the index is updated accordingly. Multiple nodes configuration is compared to the single node configuration. The average speedup achieved is $2.47\times$. This can be attributed to our use of the default text file split size of 64M. We believe that the speedup can be further improved by reducing default text file split size or alternatively, by retrieving a bigger range of blocks resulting in bigger data files. Fetching from HDFS is the typical use case where most data are already indexed

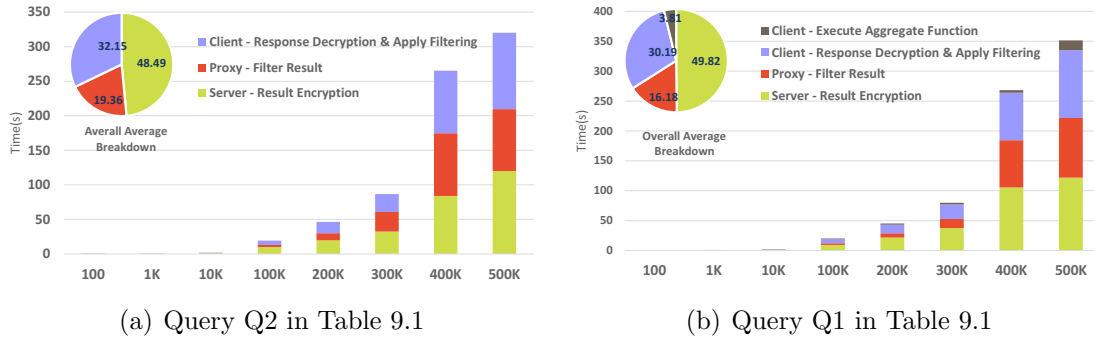


Figure 9.4: Performance breakdown following data fetch

and occasionally few blocks from live data are fetched from the Geth clients.

9.4.3 Steps following server data fetching

Following the fetch of missing/existing data, the server encrypts and computes the range hash of each batch and sends these results to the proxy, which filters the relevant batches according to the hashes it received from the client and sends them to the client. The client then decrypts the results and filters out the data that are not in the original query range (before continuing to execute the *processing query*). The summary and breakdown of the processing times for these steps are shown in Fig. 9.4 for the two types of queries in Table 9.1. Fig. 9.4(a) shows the processing times for query Q2 in Table 9.1 and Fig. 9.4(b) shows the processing times for query Q1 in Table 9.1. As can be seen, the relative processing times of the different steps are similar for 100k and the above ranges in both query types. The average breakdown of the different processing parts can be seen in the pie charts. Here, about 50% of the processing time is attributed to the results encryption by the server, followed by about 30% of the processing time for the decryption of the filtered results by the client, and about 20% of the processing time to filter the results by the proxy. In Query Q1 there is an additional use of an aggregation function, which is not specified in Q2. This adds the aggregation time to the client side, which is relatively negligible in comparison to the other parts.

Chapter 10

Conclusion and future work

In this chapter we present a conclusion, which summarizes the challenges that were considered in this work and the proposed approaches to address them. Following, we present some ideas for future work in both fields that were explored in this work.

10.1 Conclusion

Blockchain systems enable distributed, tamper-resistant, and fault-tolerant capabilities where distrusting parties can mutually manage information pseudo-anonymously through enforceable rules, in the form of transactions and contracts, and in a decentralized way. This has led various industry sectors to rapidly adopt the technology in order to manage valuable tangible and intangible assets. The promising capabilities of blockchain technology are impacted by various constraints: all blockchain data are transparent and are easily accessible to all of its users; to help protect the privacy of the blockchain users under these settings, pseudo-anonymity ensures that while the addresses and their activities are publicly available, the addresses owners cannot be easily inferred without the use of out-of-network information; all committed transactions are immutable including contract code deployment. Subsequently, no operation or deployed contract code can be modified; contracts' execution are limited by a pre-

defined gas allocation, which limits the contracts' computation capabilities at each invocation; transactions and contracts can only consume and produce blockchain data that was synchronized between all participating nodes through the consensus protocol; and in order for data to be used and the data volume is rapidly and permanently increasing. These constraints give rise to challenges that are unique to blockchain systems.

In this work we research approaches in two main fields: blockchain security and blockchain provenance management. Our proposed approaches collect and utilize comprehensive blockchain provenance to address security, defects, anomalies, and traceability issues. To empower these and other capabilities, we propose approaches to efficiently manage and query blockchain provenance.

Addressing security issues through provenance analysis in blockchains with smart contract support

As the contract applications' adoption increases, so do the applications' size and complexity; therefore, coding errors, exploit potential, and traceability requirements increase as well. Meanwhile, many blockchain platforms support automated execution of contract code. With the proliferation of such contracts, the number of incidents related to contract vulnerabilities, anomalies, and defects is also increasing, which can result in hefty financial consequences. This is mainly due to the immaturity of the field, and consequently a lack of knowledge and tools for efficient analysis and automatic verification of contracts.

In order to facilitate root-cause analysis of anomalies and investigate defects or malicious activities in contracts, the blockchain system needs to efficiently manage both data and execution flow. In Chapter 5 we propose EideticEther, a blockchain provenance collection and analysis system and explores how the system can non-intrusively capture relevant provenance information of the contracts' execution flow, their parameters, and the blockchain states before and after each contract call. We further explore how this information can be queried at different levels of granularity lev-

els to facilitate better root-cause analysis; and regulatory, quality, and maintenance management.

Existing approaches that address security issues in smart contracts employ either static analysis or dynamic analysis methods, where the dynamic analysis methods are usually resource consuming and their incorporation with existing blockchain systems is not practical. Further, these approaches offer no mitigation strategies with already deployed contracts. To address the limitations of existing approaches, in Chapter 6 we present EtherProv, which collects and tracks contract execution flow provenance by integrating static and dynamic provenance of blockchain data. EtherProv enables these capabilities through contract code instrumentation, without the need to modify the blockchain node. With our proposed efficient path profiling approach for the extraction of a contract’s executed paths, EtherProv enables the tracing of an execution flow spanning multiple interacting deployed contracts with an average instrumentation overhead of 18.9%. Moreover, EtherProv is capable of mitigating unaddressed security threats detected within already deployed contracts. Our experimental evaluation demonstrates that EtherProv is able to accurately detect several security vulnerabilities, including Liquid Ether, Re-Entrancy, and Restricted Writes, as well as analyze transaction-based security threats in deployed contracts.

The security challenges are amplified by blockchain’s pseudo-anonymity, which encodes users’ identities as public addresses. These public addresses represent the users’ accounts and cannot be easily linked back to their owners. Moreover, users are encouraged to create a new public address per transaction in order to prevent the affiliation of transactions with the same address. Current approaches to addresses affiliation use assumptions of addresses behavior that cannot be validated due to the pseudo-anonymity premise and hence the lack of ground truth datasets. In Chapter 7 we propose to leverage a stylometry approach to explore the extent to which a deployed contract’s source code can contribute to the affiliation of the deployers’

account addresses. To address this, we prepare a dataset of real-world contract data; design and implement feature selection, extraction techniques, and data refinement heuristics; and examine their effect on attribution accuracy. Our experimental results show that it is feasible to attribute Ethereum contracts to their corresponding deployers. We are able to achieve 91% accuracy in attributing source code, using less than 1% of the total features, and 80% accuracy in attributing bytecode with less than 3% of the total features. These results further demonstrate the validity of addresses affiliation using stylometry methods. In addition, we show that our approach is feasible for attribution of real-world blockchain abuse on Ponzi scheme related contracts and other real-world Ethereum scams. Furthermore, we provide an algorithm to extract distinctly contributing features from all authors and from specific authors and use it to explore our dataset and the Ponzi scheme dataset.

Efficient blockchain provenance management In Chapter 8 we propose the Authenticated Multi-Version Skip List (AMVSL) data structure to overcome the current blockchains' inherit limitations that do not support range queries or historical data management. AMVSL enables the enrichment of existing blockchain querying capabilities with multi-version range queries, while ensuring strong data authenticity and tamper-proof requirements. We demonstrate that our approach is several orders of magnitude faster than the MPT and MBT blockchain indexes currently adopted in Ethereum and Hyperledger fabric blockchains respectively.

As blockchain data consume significant space it is challenging for regulatory bodies to maintain a blockchain node for each regulated blockchain. For this reason it is important to have an efficient, secure, and remote auditing system to monitor and analyze blockchain repositories, while preserving the auditors' privacy, queries, and results. To this end, in Chapter 9 we propose a system that enables private information retrieval by utilizing cryptography techniques over semi-trusted servers and big data processing techniques, to support the above requirements. Our system provides

a secure, robust, and scalable way to process SQL queries over any blockchain. It enables multiple auditors to execute queries in an efficient and scalable way, while preserving the privacy of the auditors' identities and preventing the disclosure of the queries being used and their results. The system supports SQL queries with range and aggregate functions, which are transformed into MapReduce tasks to be run on the Hadoop system. The system uses Hadoop's MapReduce tasks to efficiently fetch missing blocks from Ethereum clients. In addition, an in-memory B^+ Tree-based index is utilized to index previously downloaded and stored Ethereum blocks. We conducted a systematic performance evaluation, which demonstrates the system's performance improvement, which correlates with the number of Hadoop nodes and synchronized Ethereum clients.

10.2 Future work

In this section we discuss possible directions for future work.

Cross-blockchain attacks As the blockchain ecosystem evolves so do its security concerns. A natural and crucial progression of blockchain applicability involves cross-blockchain interoperability [111] where a transaction can span multiple blockchains. Bridges, e.g., Polkadot [40], are used to synchronize transactions between blockchains. A bridge uses different types of contracts to interface with the different blockchains it bridges. As each blockchain and its contracts are susceptible to their own attacks and vulnerabilities, a bridge needs to account for the security issues of all blockchains and contracts it interfaces with. Since security issues are rapidly mounting in each blockchain, accounting for all security issues across all interfaced blockchains is challenging. Further, since bridges require a large amount of gas, they are typically used to transfer a large amount of money between blockchains. These two factors can contribute to large losses as a result of attacks. Indeed, in 2022

alone, 13 separate bridge attacks resulted in a loss of \$2 billion USD [28]. Hence, an efficient analysis, detection, and mitigation of such attacks is crucial. Our proposed EtherProv framework (Section 6) can be extended to different contract languages, which can enable the analysis and mitigation of attacks in individual bridge interfaces. As EtherProv operates on abstracted CFGs that are detached from a specific contract language, it can connect the various CFGs of the interfaced blockchain contracts to an extended cross-blockchain CFG, where the specific operations in each node are provided in a unified language. This will enable to analyze and possibly mitigate cross-blockchain attacks.

Account address affiliation In chapter 7 we explored the extent to which unique features of source code and bytecode of Ethereum contracts can represent the coding style of contract developers and enable affiliating their deployers' addresses. We achieve accuracy of 91% for source codes and 80% for bytecodes using only a fraction of available features. Exploring attribution of contracts on a larger scale will potentially provide insight into the behavior of contracts' authors across accounts and is likely to reveal suspicious tendencies. In our experiments we used a supervised machine learning approach. Since extracting authors' identity is not possible as part of the blockchain's premise, exploring an unsupervised approach's validity can prove more applicable for real-world needs as it will enable "open set" recognition of new authors not previously seen by the model.

Anomaly detection using machine learning Detecting contracts' attacks based on detailed execution flow provenance requires knowing the attack patterns in advance. Instead, a more general approach that is based on anomaly detection can be applied. A machine learning model can be trained on the available transaction features and possibly additional features to enable the anomaly prediction. The feasibility of this approach can be explored by experimenting with the various features,

and examining the prediction results as they relate to known or unknown attacks.

Multi version indexes using light clients Our AMVSL index, introduced in Chapter 8, enables an efficient management of authenticated multi version data. However, storing historical data in each blockchain node can get expensive. Light clients enable querying blockchain data, without storing the entire data on the light node. This is done by querying full nodes that keep a copy of all blockchain data, and authenticating the retrieved data. Extending the AMVSL index to enable authenticated queries will help alleviate the high storage requirements by enabling light clients to query historical data from full nodes.

10.3 Closing thoughts

In this work, we have presented approaches to address challenges in two main fields: blockchain security and blockchain provenance management. We have proposed systems and methods that collect and utilize comprehensive blockchain provenance to address security issues, defects, anomalies, and traceability issues, and to efficiently manage and query blockchain provenance.

The field of blockchain technology is still in its infancy, and there are many challenges that need to be addressed. The challenges we addressed in this work are just a small subset of the challenges that the community faces.

As blockchain technology continues to evolve and mature, we expect to see many new applications and use cases. We also expect to see new challenges arise that will need to be addressed. The approaches we have presented in this work provide a basis for addressing these challenges and for enabling the realization of the full potential of blockchain technology.

In conclusion, we believe that the research presented in this work has made important contributions to the field of blockchain technology. We hope that our work will inspire

further research and development in this exciting and rapidly evolving field.

References

- [1] *An In-Depth Look at the Parity Multisig Bug.*
<https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/>.
- [2] *cardano.* <https://cardano.org/>.
- [3] *eos.io.* <https://eos.io/>.
- [4] *Ganache.* <https://www.trufflesuite.com/ganache>.
- [5] *gartner blockchain-platforms.*
<https://www.gartner.com/reviews/market/blockchain-platforms>.
- [6] *Go ethereum.* <https://geth.ethereum.org>.
- [7] *hyperledger fabric.* <https://github.com/hyperledger/fabric>.
- [8] *king of the ether.* <https://www.kingoftheether.com/postmortem.html>.
- [9] *King of the ether throne.* <https://github.com/kieranelby/kingoftheetherthrone>.
- [10] *Neo4j.* <https://neo4j.com/>.
- [11] *Solidity library calls.*
<https://solidity.readthedocs.io/en/develop/contracts.html#libraries>.
- [12] *Apache hadoop.* <https://hadoop.apache.org/>, 2009.
- [13] *The DAO attacked: Code Issue Leads to 60 Million Ether Theft.*, 2016.

- [14] *Parity security alert*. <https://www.parity.io/security-alert-2/>, 2017.
- [15] *Blockchain applications for the modern nation*. <https://cryptodaily.co.uk/2018/03/blockchain-applications/>, 2018.
- [16] *Blockchain has grabbed the attention of investors*. <https://www.cnbc.com/2018/04/02/blockchain-has-garbled-the-attention-of-investors.html>, 2018.
- [17] *Coinmarketcap*. <https://coinmarketcap.com/>, 2018.
- [18] *How ibm blockchain can improve government services and ensure trust*. <https://www.ibm.com/downloads/cas/2vnrqx9v>, 2018.
- [19] *Three near-term applications for blockchain technology*. <https://www.forbes.com/sites/forbesfinancecouncil/2018/03/28/three-near-term-applications-for-blockchain-technology/2/#6c9f49c6310d>, 2018.
- [20] *Solidity documentation v0.5.10*. <https://solidity.readthedocs.io/en/v0.5.10/>, 2019.
- [21] *Mythril classic*. <https://github.com/consensys/mythril-classic>, 2021.
- [22] *Etherscamdb*, <https://etherscamdb.info/>.
- [23] *etherscan.io*, <https://etherscan.io/>.
- [24] *Porosity*, <https://github.com/comaeio/porosity>.
- [25] *Radare2*, <https://github.com/radare/radare2>.
- [26] *Datasets used in this paper*, <https://github.com/shomzy/Lino1910>.
- [27] *Scikit-learn decision trees complexity*, <https://scikit-learn.org/stable/modules/tree.html#complexity>.

- [28] *\$2B in crypto stolen from cross-chain bridges this year: Chainalysis*, <https://cointelegraph.com/news/2b-in-crypto-stolen-from-cross-chain-bridges-this-year-chainalysis>, 2022.
- [29] *A disastrous vulnerability found in smart contracts of BeautyChain (BEC)*, <https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e>.
- [30] Sadia Afroz, Aylin Caliskan Islam, Ariel Stolerman, Rachel Greenstadt, and Damon McCoy, *Doppelgänger finder: Taking stylometry to the underground*, 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 212–226.
- [31] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al., *Hyperledger fabric: a distributed operating system for permissioned blockchains*, EuroSys, 2018.
- [32] *Antlr*, <http://wwwantlr.org/>, 1995.
- [33] Thomas Ball, *Efficiently counting program events with support for on-line queries*, ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994), no. 5, 1399–1410.
- [34] Thomas Ball and James R Larus, *Efficient path profiling*, MICRO, IEEE, 1996, pp. 46–57.
- [35] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia, *Dissecting ponzi schemes on ethereum: identification, analysis, and impact*, Future Generation Computer Systems **102** (2020), 259–277.
- [36] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov, *Deanonymisation of clients in bitcoin p2p network*, SIGSAC CCS, ACM, 2014, pp. 15–29.

- [37] Leo Breiman, *Classification and regression trees*, Routledge, 2017.
- [38] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz, *Vandal: A scalable security analysis framework for smart contracts*, arXiv preprint arXiv:1809.03981 (2018).
- [39] Ethan Buchman, *Tendermint: Byzantine fault tolerance in the age of blockchains*, Ph.D. thesis, University of Guelph, 2016.
- [40] Jeff Burdges, Alfonso Cevallos, Peter Czaban, Rob Habermeier, Syed Hosseini, Fabio Lama, Handan Kilinc Alper, Ximin Luo, Fatemeh Shirazi, Alistair Stewart, et al., *Overview of polkadot and its design considerations*, arXiv preprint arXiv:2005.13456 (2020).
- [41] Steven Burrows, Alexandra L Uitzenbogerd, and Andrew Turpin, *Comparing techniques for authorship attribution of source code*, spe **44** (2014), no. 1, 1–32.
- [42] V. Buterin, *Eip-170*. <https://github.com/ethereum/eips/blob/master/eips/eip-170.md>, 2016.
- [43] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt, *De-anonymizing programmers via code stylometry*, 24th USENIX, 2015, pp. 255–270.
- [44] Wren Chan and Aspen Olmsted, *Ethereum transaction graph analysis*, 12th ICITST, IEEE, 2017, pp. 498–500.
- [45] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al., *SODA: A generic online detection framework for smart contracts*, NDSS, 2020.

- [46] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, et al., *Dataether: Data exploration framework for ethereum*, ICDCS, 2019, pp. 1369–1380.
- [47] James Cheney, Laura Chiticariu, and Wang-Chiew Tan, *Provenance in databases: Why, how, and where*, Now Publishers Inc, 2009.
- [48] Mauro Conti, E Sandeep Kumar, Chhagan Lal, and Sushmita Ruj, *A survey on security and privacy issues of bitcoin*, IEEE CST **20** (2018), no. 4, 3416–3452.
- [49] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt, *Poster: Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments*, IEEE/ACM 40th ICSE-Companion, 2018, pp. 356–357.
- [50] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen, *Eidetic systems*, 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), 2014, pp. 525–540.
- [51] Vikram Dhillon, David Metcalf, and Max Hooper, *The hyperledger project*, Blockchain enabled applications, Springer, 2017, pp. 139–149.
- [52] Yi Ding, Chenshuo Wang, Qionghui Zhong, Haisheng Li, Jinjing Tan, and Jie Li, *Function-level dynamic monitoring and analysis system for smart contract*, IEEE Access (2020).
- [53] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan, *Blockbench: A framework for analyzing private blockchains*, SIGMOD, 2017.
- [54] Jules DuPont and Anna Cinzia Squicciarini, *Toward de-anonymizing bitcoin by mapping users location*, CODASPY, ACM, 2015, pp. 139–141.
- [55] *Erc20exporter*, <https://github.com/gobitfly/erc20-explorer>, 2017.

- [56] *Erc20 token standard*, https://theethereum.wiki/w/index.php/ERC20-Token_Standard#The_ERC20-Token_Standard_Interface.
- [57] *Etcexplorer*, <https://github.com/ethereumproject/explorer>, 2016.
- [58] *Etherchain light*, <https://github.com/gobitfly/etherchain-light>, 2017.
- [59] *Ethereum explorer*, <https://github.com/mix-blockchain/ethereum-explorer>, 2017.
- [60] *Ethereum javascript api*, <https://web3js.readthedocs.io/>, 2014.
- [61] *Ethereum json rpc*, <https://github.com/ethereum/wiki/wiki/JSON-RPC>, 2014.
- [62] *Ethereumscraper*, <https://github.com/medvedev1088/ethereum-scraper>, 2018.
- [63] *Ethexplorer*, <https://github.com/etherparty/explorer>, 2015.
- [64] ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued, *Ethnews: Hkg token has a bug and needs to be reissued (2017).*, (2017).
- [65] *Ethplorer*, <https://ethplorer.io/>, 2016.
- [66] Josselin Feist, Gustavo Greico, and Alex Groce, *Slither: A static analysis framework for smart contracts*, IEEE Press, 2019.
- [67] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis, *Examining the significance of high-level programming features in source code author classification*, Journal of Systems and Software **81** (2008), no. 3, 447–460.
- [68] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E Chaski, and Blake Stephen Howald, *Identifying authorship by byte-level n-*

- grams: The source code author profile (scap) method*, IJDE **6** (2007), no. 1, 1–18.
- [69] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas, *Effective identification of source code authors using byte-level information*, ICSE, ACM, 2006, pp. 893–896.
- [70] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas, *Source code author identification based on n-gram author profiles*, AIAI (2006), 508–515.
- [71] *From farm to blockchain: Walmart tracks its lettuce*, <https://www.nytimes.com/2018/09/24/business/walmart-blockchain-lettuce.html>.
- [72] Valentina Gatteschi, Fabrizio Lamberti, Claudio Demartini, Chiara Pranteda, and Víctor Santamaría, *Blockchain and smart contracts for insurance: Is the technology mature enough?*, Future Internet **10** (2018), no. 2, 20.
- [73] Asem Ghaleb and Karthik Pattabiraman, *How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection*, Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (New York, NY, USA), ISSTA 2020, Association for Computing Machinery, 2020, p. 415–427.
- [74] Boris Glavic and Gustavo Alonso, *Perm: Processing provenance and data on the same data model through query rewriting*, 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 174–185.
- [75] LM Goodman, *Tezos: A self-amending crypto-ledger position paper*, Aug **3** (2014), 2014.

- [76] Michael T Goodrich and Roberto Tamassia, *Efficient authenticated dictionaries with skip lists and commutative hashing*, August 14 2007, US Patent 7,257,711.
- [77] Shelly Grossman et al., *Online detection of effectively callback free objects with applications to smart contracts*, Proc. ACM Program. Lang. **2** (2017).
- [78] Ákos Hajdu and Dejan Jovanović, *solc-verify: A modular verifier for solidity smart contracts*, Verified Software. Theories, Tools, and Experiments (Cham) (Supratik Chakraborty and Jorge A. Navas, eds.), Springer International Publishing, 2020, pp. 161–179.
- [79] Matthew B Hoy, *An introduction to the blockchain and its implications for libraries and medicine*, Medical reference services quarterly **36**.
- [80] Herbert Jordan, Bernhard Scholz, and Pavle Subotić, *Soufflé: On synthesis of program analyzers*, CAV, 2016, pp. 422–430.
- [81] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina, *Code authorship attribution: Methods and challenges*, CSUR **52** (2019), no. 1, 3.
- [82] Vaibhavi Kalgutkar, Natalia Stakhanova, Paul Cook, and Alina Matyukhina, *Android authorship attribution through string analysis*, Proceedings of the 13th International Conference on Availability, Reliability and Security (New York, NY, USA), ARES 2018, Association for Computing Machinery, 2018.
- [83] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma, *ZEUS: analyzing safety of smart contracts*, NDSS, 2018.
- [84] D Kaminsky, *Black ops of tcp/ip*, Black Hat USA (2011), 44.
- [85] Robin Klusman and Tim Dijkhuizen, *Deanonymisation in ethereum using existing methods for bitcoin*, (2018).

- [86] Jesse Kornblum, *Identifying almost identical files using context triggered piecewise hashing*, Digital investigation **3** (2006), 91–97.
- [87] Philip Koshy, Diana Koshy, and Patrick McDaniel, *An analysis of anonymity in bitcoin using p2p network traffic*, ADCS, Springer, 2014, pp. 469–485.
- [88] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis, *A probabilistic approach to source code authorship identification*, ICIT, IEEE, April 2007, pp. 243–248.
- [89] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O’Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai, *Roaring bitmaps: Implementation of an optimized software library*, Software: Practice and Experience **48** (2018), no. 4, 867–895.
- [90] *Leveldb*, <https://dbdb.io/db/leveldb>, 2014.
- [91] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin, *Dynamic authenticated index structures for outsourced databases*, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 2006, pp. 121–132.
- [92] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen, *A survey on the security of blockchain systems*, FGCS (2017).
- [93] Yang Li, Kai Zheng, Ying Yan, Qi Liu, and Xiaofang Zhou, *EtherQL: a query layer for blockchain system*, DASFAA, 2017.
- [94] Xueping Liang, Sachin Shetty, Deepak Tosh, Charles Kamhoua, Kevin Kwiat, and Laurent Njilla, *Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability*, 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE, 2017, pp. 468–477.

- [95] Shlomi Linoy, Suprio Ray, and Natalia Stakhanova, *Etherprov: provenance-aware detection, analysis, and mitigation of ethereum smart contract security issues*, IEEE Blockchain, IEEE, 2021, pp. 1–10.
- [96] Yang Liu, Debiao He, Mohammad S Obaidat, Neeraj Kumar, Muhammad Khurram Khan, and Kim-Kwang Raymond Choo, *Blockchain-based identity management systems: A review*, Journal of network and computer applications **166** (2020), 102731.
- [97] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor, *Making smart contracts smarter*, 2016.
- [98] Fuchen Ma, Ying Fu, Meng Ren, Mingzhe Wang, Yu Jiang, Kaixiang Zhang, Huizhong Li, and Xiang Shi, *Evm*: from offline detection to online reinforcement for ethereum virtual machine*, SANER, IEEE, 2019, pp. 554–558.
- [99] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage, *A fistful of bitcoins: characterizing payments among men with no names*, IMC, ACM, 2013, pp. 127–140.
- [100] Ralph Merkle, *Merkle tree patent*, Ralph Merkle (1979).
- [101] Merkle Patricia Trie Java implementation, <https://github.com/serdaroquai/patricia-merkle-trie>, 2019.
- [102] Matthias Mettler, *Blockchain technology in healthcare: The revolution starts here*, IEEE Healthcome, 2016.
- [103] Hui Miao, Amit Chavan, and Amol Deshpande, *ProvdB: Lifecycle management of collaborative analysis workflows*, Proceedings of the 2nd Workshop on Human-in-the-Loop Data Analytics, 2017, pp. 1–6.
- [104] Satoshi Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, (2008).

- [105] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, Irfan Awan, and Andrea Cullen, *Automated labeling of unknown contracts in ethereum*, 26th ICCCN, IEEE, 2017, pp. 1–6.
- [106] Wellington Oliveira, Daniel De Oliveira, and Vanessa Braganholo, *Provenance analytics for workflow-based computational experiments: A survey*, ACM Computing Surveys (CSUR) **51** (2018), no. 3, 1–25.
- [107] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al., *Scikit-learn: Machine learning in python*, Journal of machine learning research **12** (2011), no. Oct, 2825–2830.
- [108] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song, *FalconDB: Blockchain-Based Collaborative Database*, SIGMOD, 2020, p. 637–652.
- [109] Daniel Perez and Ben Livshits, *Smart contract vulnerabilities: Vulnerable does not imply exploited*, 30th USENIX Security Symposium (USENIX Security 21), 2021.
- [110] Deepak Puthal, Nisha Malik, Saraju P Mohanty, Elias Kougianos, and Gautam Das, *Everything you wanted to know about the blockchain: Its promise, components, processes, and problems*, IEEE Consumer Electronics Magazine **7** (2018).
- [111] Ilham A Qasse, Manar Abu Talib, and Qassim Nasir, *Inter blockchain communication: A survey*, Proceedings of the ArabWIC 6th Annual International Conference Research Track, 2019, pp. 1–6.
- [112] Aravind Ramachandran and Murat Kantarcioglu, *Smartprovenance: a distributed, blockchain based dataprovenance system*, Proceedings of the Eighth

- ACM Conference on Data and Application Security and Privacy, 2018, pp. 35–42.
- [113] Fergal Reid and Martin Harrigan, *An analysis of anonymity in the bitcoin system*, Security and privacy in social networks, Springer, 2013, pp. 197–223.
- [114] *Rocksdb*, <http://rocksdb.org/>, 2013.
- [115] M. Rodler, Wenting Li, G. Karame, and L. Davi, *Sereum: Protecting existing smart contracts against re-entrancy attacks*, NDSS (2019).
- [116] Dorit Ron and Adi Shamir, *Quantitative analysis of the full bitcoin transaction graph*, ADCS, Springer, 2013, pp. 6–24.
- [117] Nathan Rosenblum, Xiaojin Zhu, and Barton P Miller, *Who wrote this code? identifying the authors of program binaries*, ESORICS, Springer, 2011, pp. 172–189.
- [118] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang, *Fine-grained, secure and efficient data provenance on blockchain systems*, Proceedings of the VLDB Endowment **12** (2019), no. 9, 975–988.
- [119] Marc Santamaria Ortega, *The bitcoin transaction graph anonymity*, (2013).
- [120] Upendra Sapkota, Steven Bethard, Manuel Montes, and Tamar Solorio, *Not all character n-grams are created equal: A study in authorship attribution*, Proceedings of the 2015 conference of the North American chapter of the association for computational linguistics: Human language technologies, 2015, pp. 93–102.

- [121] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich, *Blurring the Lines between Blockchains and Database Systems: The Case of Hyperledger Fabric*, SIGMOD, 2019, p. 105–122.
- [122] Maxim Shevertalov, Jay Kothari, Edward Stehle, and Spiros Mancoridis, *On the use of discretized source code metrics for author identification*, SSBSE, IEEE, May 2009, pp. 69–78.
- [123] Grigori Sidorov, Francisco Velasquez, Efstathios Stamatatos, Alexander Gelbukh, and Liliana Chanona-Hernández, *Syntactic n-grams as machine learning features for natural language processing*, Expert Systems with Applications **41** (2014), no. 3, 853–860.
- [124] Lucy Simko, Luke Zettlemoyer, and Tadayoshi Kohno, *Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution*, PoPETs (2018), no. 1, 127 – 144.
- [125] Tom Simonite, *\$80 million hack shows the dangers of programmable money*, Connectivity, June **17** (2016), 2016.
- [126] *Slither detectable contract vulnerabilities*, <https://github.com/crytic/slither#detectors>.
- [127] Michele Spagnuolo, Federico Maggi, and Stefano Zanero, *Bitiodine: Extracting intelligence from the bitcoin network*, ADCS, Springer, 2014, pp. 457–468.
- [128] *How do supply chains use Big Data?*, <https://www.supplychaindive.com/news/how-big-data-application-supply-chain-Deloitte-digital-stack/435866/>, 2017.
- [129] Matthew F. Tennyson and Francisco J. Mitropoulos, *Choosing a profile length in the scap method of source code authorship attribution*, SoutheastCon, IEEE, March 2014, pp. 1–6.

- [130] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov, *Smartcheck: Static analysis of ethereum smart contracts*, WETSEB, 2018, pp. 9–16.
- [131] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State, *The eye of horus: Spotting and analyzing attacks on ethereum smart contracts*, arXiv preprint arXiv:2101.06204 (2021).
- [132] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev, *Securify: Practical security analysis of smart contracts*, CCS, 2018, pp. 67–82.
- [133] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev, *Securify: Practical security analysis of smart contracts*, CCS (New York, NY, USA), Association for Computing Machinery, 2018, p. 67–82.
- [134] Xinming Wang, Jiahao He, Zhijian Xie, Gansen Zhao, and Shing-Chi Cheung, *ContractGuard: Defend ethereum smart contracts with embedded intrusion detection*, IEEE TSC (2019), 314–328.
- [135] Daniel Watson, *Source code stylometry and authorship attribution for open source*, Master’s thesis, University of Waterloo, 2019.
- [136] *Web3j*, <https://docs.web3j.io/>, 2016.
- [137] Gavin Wood et al., *Ethereum: A secure decentralised generalised transaction ledger*, Ethereum project yellow paper (2014), 1–32.
- [138] Cheng Xu, Ce Zhang, and Jianliang Xu, *vchain: Enabling verifiable boolean range queries over blockchain databases*, SIGMOD, 2019.

- [139] Zihuan Xu, Siyuan Han, and Lei Chen, *Cub, a consensus unit-based storage scheme for blockchain system*, ICDE ,2018.
- [140] Emre Yavuz, Ali Kaan Koç, Umut Can Çabuk, and Gökhan Dalkılıç, *Towards secure e-voting using ethereum blockchain*, 2018 6th ISDFS.
- [141] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao, *Analysis of indexing structures for immutable data*, Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, 2020, pp. 925–935.
- [142] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong, *Spitz: A Verifiable Database System*, Proc. VLDB Endow. **13** (2020), no. 12, 3449–3460.
- [143] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin, *TXSPECTOR: Uncovering attacks in ethereum from transactions*, USENIX Security 20, 2020, pp. 2775–2792.
- [144] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou, *IntegriDB: Verifiable SQL for Outsourced Databases*, CCS, 2015, p. 1480–1491.
- [145] Yanchao Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Gang Qin, and Yingjie Yang, *Towards rich query blockchain database*, CIKM, 2020, p. 3497–3500.

Appendix A

A.1 Eidetic blockchain frameworks with Enhanced Provenance

A.1.1 Retail example implementation details

Contract	Function	Description
sales	suppliers_request	Receives a product_id and forwards it to the suppliers' contract by calling the suppliers_request function. Returns the list of suppliers with their product price and quantity.
	order_request	Records the order details in the client_orders_DB and calls the order_request function at the suppliers' contract. The response should contain a shipment_id, which is updated in the order details.
	shipping_confirmation	Updates its product order's shipment_confirmation_date.
	process_payment	Receives and updates the payment confirmation from the client through a payment_id. Issues a payment to the supplier and sends the payment confirmation id by calling the suppliers contract's process_payment function.
suppliers	suppliers_request	Receives a product_id and returns the list of suppliers with their product price and quantity.
	order_request	Records the order details in the sales_orders_DB and calls the shipping_request function at the shipping contract. The response should contain a shipment_id, which is updated in the order details and returned to the calling contract.
	shipping_confirmation	Updates its product shipment_confirmation_date and calls the shipment_confirmation function at the suppliers' contract.
	process_payment	Receives and updates the payment confirmation received from sales through a payment_id. Issues a payment for the shipping and sends the payment confirmation id to the shipping contract's process_payment function.
shipping	shipping_request	The client physical address details are updated, a shipment_id is generated and returned to the calling contract.
	shipping_confirmation	Upon receiving a confirmation of the product delivery, updates its product's shipment_confirmation_date and calls the shipment_confirmation function at the suppliers' contract.
	process_payment	Receives and updates the payment confirmation from the client through a payment_id.

Table A.1: Retail - contracts' API description

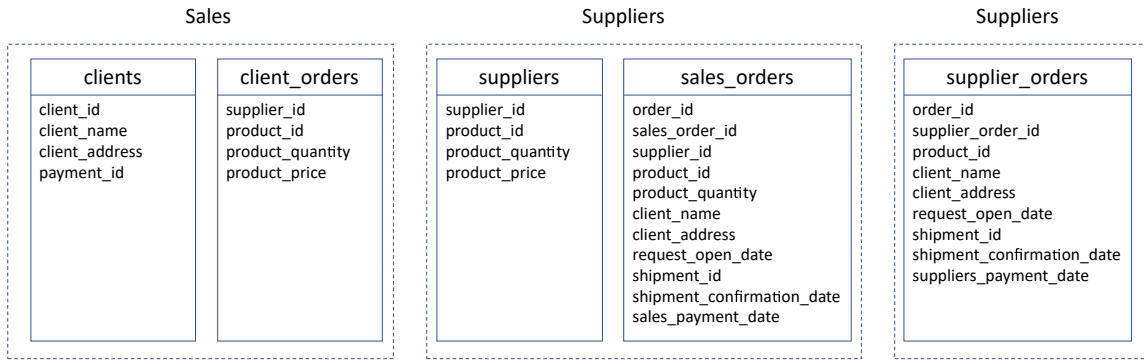


Figure A.1: Database schemes for all departments

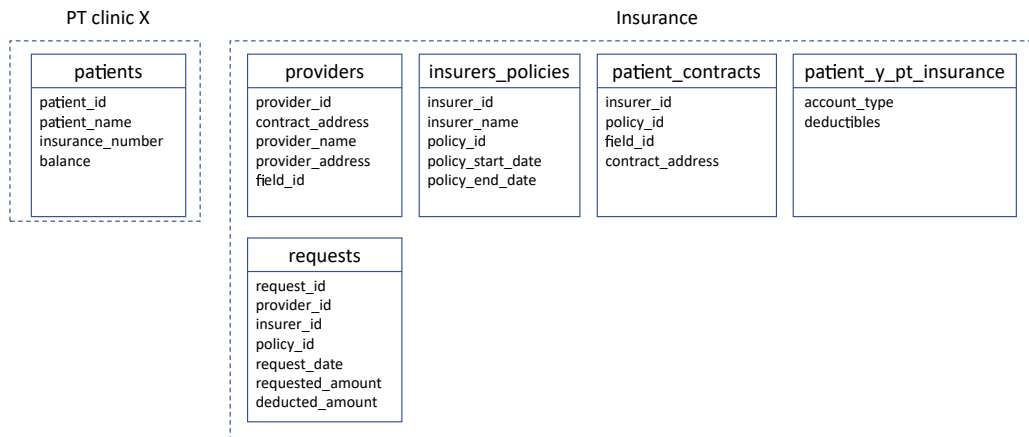


Figure A.2: Health insurance scenario - database schema details

Departments' contracts API overview Table A.1 provides an overview of the contracts' API used in the retail scenario, for all departments.

Database schemes for all departments Fig. A.1 provides the model schema for all departments contracts.

A.1.2 Health insurance example implementation details

Contracts' API overview Table A.2 provides an overview of the contracts' API in the health insurance scenario.

Database schemes Fig. A.2 presents the database schema used in the health insurance scenario.

Contract	Function	Description
PT clinic X	register_insurance_request	Calls the Insurance contract's register_provider_request command with its details such as name, physical address, and field of service.
	claim_request	Receives the patient details and forwards it to the insurance contract using the claim_request command. Returns the receipt including the deducted amount.
Insurance	register_insurer	Receives the insurer details and updates the insurers_policies_DB accordingly. For each service benefit in the policy creates a service benefits contract, e.g., patient.y.pt.insurance for the PT service with the corresponding initial values, per the insurance policy. Updates the patient_contracts_DB with the contract_address of the created service benefits contract.
	register_provider_request	Receives the providers details and updates its providers_DB. The provider's contract address is extracted from the contract call and is saved to the contract_address field.
	claim_request	Receives the patient details and the provider's contract address (from the contract call). Locates the provider details to ensure the provider is registered in the providers_DB. Opens a request and stores it in the requests_DB. Using the received patient_id, policy_id, and field_id it extracts the contract_address from the patient_contracts_DB. The contract_address is used to call the patient.y.pt.insurance contract with the amount to deduct. Returns the receipt including the deducted amount.
patient.y.pt.insurance	create	Creates the contact with the provided initial account_type and deductibles.
	process_request	Receives the amount to deduct, deducts the maximum available amount, and returns the actual deducted amount.

Table A.2: Health insurance - contracts' API description

Appendix B

B.1 The EtherProv framework

This section provides a more detailed overview of the EtherProv model to better explain the collected static and dynamic provenance.

```
1 static_contract(batch, contract, contract_name, is_signature_only)
2
3 static_contract_state_param(batch, contract_state_param, static_contract, solidity_type, name,
   initial_value)
4
5 static_fun(batch, fun, static_contract, name, type, static_entry_node, is_payable, visibility)
6
7 static_fun_param(batch, fun_param, static_fun, order_index, solidity_type, name)
8
9 static_node(batch, node, static_fun, type, expression)
10
11 static_var(batch, var, static_contract, static_fun, name, solidity_type, is_storage, is_const,
   visibility, expression)
12
13 static_edge(batch, edge, from_static_node, to_static_node)
14
15 static_ssa_edge(batch, ssa_edge, from_static_ssa_node, to_static_ssa_node, is_to_call_site,
   related_to_call_site_edge)
16
17 static_path(batch, path, static_edge)
18
19 static_path_first_read_last_written_state_param(batch,
   path_first_read_last_written_state_param, static_path, static_contract_state_param,
   first_read_or_last_written)
20
```

```

21 static_ssa_node(batch, ssa_node, static_node, type, expression, call_value, is_true_branch,
    is_false_branch)
22
23 static_ssa_node_var(batch, ssa_node_var, static_node_var, name, solidity_type, expression,
    points_to, is_storage, is_const, visibility)
24
25 static_ssa_node_operation_with_l_value(batch, static_ssa_node, lvalue_static_ssa_node_var)
26
27 static_ssa_node_index(batch, static_ssa_node, var_left_static_ssa_node_var,
    var_right_static_ssa_node_var)
28
29 static_ssa_non_contract_fun_call(batch, static_ssa_node, name)
30
31 static_ssa_fun_return_var(batch, static_ssa_node, static_ssa_node_var, order)
32
33 static_ssa_node_read_var(batch, static_ssa_node, static_ssa_node_var, order)
34
35 dynamic_contract(batch, contract, deployer_address, initial_ether, contract_address,
    contract_name)
36
37 dynamic_sc_call(sc_call, dynamic_contract, block, transaction, caller_address, ether_start,
    ether_end, static_fun)
38
39 dynamic_sc_fun_call_param(dynamic_sc_call, static_fun_param, value)
40
41 dynamic_path(dynamic_sc_call, static_path, order, path_count)
42
43 dynamic_sc_call_state_param_written(dynamic_sc_call_state_param_written, dynamic_sc_call,
    static_contract_state_param, value, prev_dynamic_sc_call_state_param_written)
44
45 dynamic_sc_call_state_param_read(dynamic_sc_call, static_contract_state_param,
    dynamic_sc_call_state_param_written)

```

Listing B.1: EtherProv model

Listing. B.1 presents the complete EtherProv model, also referred to as the extensional database which is comprised to Datalog fact tables (or relations) that manage the static provenance (prefixed with `static`) and relations that manage the dynamic provenance (prefixed with `dynamic`). We next discuss the static and dynamic relations in more detail.

Static provenance EtherProv collects the following general contract static data:

- **static_contract** - contract name and unique identifier.
- **static_contract_state_param** - contract storage state parameter data such as type, initial value, etc.
- **static_fun** - contract function data such as the function's owning contract id, is the function payable, function visibility, etc.
- **static_fun_param** - function arguments data such as name, type, parameter index, etc.

EtherProv collects the following SSA node static data:

- **static_ssa_node** - an SSA node contain information on the SSA expression, i.e. assignment, function call, condition, etc.
- **static_ssa_edge** - contains the edges that connect SSA nodes in each function as represented in the CFG IR. Each edge contains information on the from/to SSA node ids, and if the SSA node is a call site.

Each SSA node contains different types of expressions such as variable initialization, assignment, condition, function call, etc. EtherProv collects such information for each SSA node, according to its expression type:

- **static_ssa_node_var** - contains information on a variable used in an SSA node such as its name, type, if stored in storage, is constant, its related non-SSA node, etc.
- **static_ssa_node_read_var** - contains the connection between an SSA node and the variable that is read in it.

- **static_ssa_node_operation_with_l_value** - contains the connection between an SSA node, which is an assignment expression, and the variable id of the left value in the assignment expression.
- **static_ssa_node_index** - contains the connection between an SSA node, which contains an index expression (such as in referencing an array cell), the variable id of the indexed variable, and the variable id of the variable used as the index.
- **static_ssa_fun_return_var** - contains the connection between an SSA node, which contains a function call, and the variable id of the variable used to store a function's return value.
- **static_ssa_non_contract_fun_call** - contains information of Solidity low level functions such as "delegatecall". When a SSA node is a call site, the called function can be a contract function (internal or to a function in a different contract) or a Solidity low level function.

The following data are collected in non-SSA form:

- **static_node** - contains a non-SSA node id, the function id to which it is related, the node's type, and the non-SSA expression.
- **static_edge** - contains the edges that connect non-SSA nodes in each function as represented in the CFG IR. Each edge contains information on the from/to non-SSA node ids.
- **static_var** - connects between the non-SSA node and the variable used by it, the function id to which it is related, the node's Solidity type, and the non-SSA expression it is related to.

The new edges that are added from/to each SSA node, if it is a call site are added to the **static_ssa_edge** schema. Similarly, the non-SSA new edges are added to the **static_edge** schema.

All CFG possible paths in and between contracts and their encoding are stored in the **static_path** schema. Each instrumented CFG node contains the Solidity code statement, its row/column location in the source code, and the related instrumentation. The instrumented CFG is then used to instrument the Solidity source code, as discussed next.

Dynamic provenance EtherProv store the dynamic data in the following facts:

- **dynamic_contract** - records a deployed contract data such as the related static contract id, deployer address, initial ether and the deployed contract's address.
- **dynamic_smart_contract_call** - contains the unique id of the contract call and the contract id, the contract function id called, the block id, transaction id, caller address, and the ether amount at the end of the call.
- **dynamic_smart_contract_fun_call_param** - contains data of the parameters and their values used to call the contract function.

EtherProv stores all the paths that are related to a specific contract and the number of consecutive times each path part was executed to **dynamic_path**. When querying the transaction's encoded path, all encoded path parts can be decoded to the relevant CFG edges using the static data Datalog fact **static_path**, which contains a mapping between each encoded path part to its corresponding CFG edges. **static_edge** and **static_node** are used to extract static information related to each edge including the statements' text.

To save computation and space, EtherProv collects only changed storage states for a specified contract execution. To do so, when a static path and its encoding are extracted, EtherProv extrapolates which states are written-to in a specific path using static analysis, and stores it in the **static_path_first_read_last_written_state_param** Datalog fact. EtherProv queries this fact and collects only the states that

were changed by the execution. EtherProv extracts the following dynamic contract data after each contract call:

- **dynamic_smart_contract_call_state_param_written** - the storage state parameters' values that were written-to, in an execution path issued by a specific contract call.
- **dynamic_smart_contract_call_state_param_read** - the storage state parameters that were read-from before any changes were made by an execution path issued by a specific contract call.

Appendix C

C.1 Caliskan source code feature set

The Caliskan et al. feature set was originally developed for C and C++ programs. We mapped the selected features to the corresponding Solidity features.

C.1.1 Lexical features

1. $WordUnigramTF$ - Term frequency of word unigrams in source code (split is done on spaces and added spaces as well).
2. $\ln(Numkeyword/FileLength)$ - Log of the number of occurrences of keyword divided by file length in characters, where keyword is one of: do, elseif, if, else, for or while. Since switch is associated only with assembly it was not used in this implementation.
3. $\ln(numTernary/FileLength)$ - Log of the number of ternary operators divided by file length in characters.
4. $\ln(numTokens/FileLength)$ - Log of the number of word tokens divided by file length in characters. Tokens include: Pragma Name, Import directive alias, Import directive name, contract/interface/library name, State variable name, Using package name, Struct name, Modifier name, Modifier invocation name, Function name,

Event name, Enum value, Parameter name, Event parameter, name, Variable name, User defined type name, Name in name-value pair, Assembly item, Assembly call, Assembly identifier, Assembly stack, assignment value, Label name, Assembly function name, Sub assembly name, and other Literals like booleans, hex values etc.

5. $\ln(\text{numComments}/\text{FileLength})$ - Log of the number of comments divided by file length in characters.

6. $\ln(\text{numLiterals}/\text{FileLength})$ - Log of the number of strings, characters, and numeric literals divided by file length in characters: Version literal, String literal, Boolean literal, Number literal, Hex literal, and Assembly literal.

7. $\ln(\text{numKeywords}/\text{FileLength})$ - Log of the number of unique keywords used divided by file length in characters. Unique keywords include: If, else, else-if, while, for, do, while, new, from, import, pragma, as, contract, interface, library, is, using, struct, constructor, modifier, function, returns, event, anonymous, enum, mapping, memory, storage, continue, break, return, throw, emit, var, uint, uint8, uint16, uint24, uint32, uint40, uint48, uint56, uint64, uint72, uint80, uint88, uint96, uint104, uint112, uint120, uint128, uint136, uint144, uint152, uint160, uint168, uint176, uint184, uint192, uint200, uint208, uint216, uint224, uint232, uint240, uint248, uint256, Byte, bytes, bytes1, bytes2, bytes3, bytes4, bytes5, bytes6, bytes7, bytes8, bytes9, bytes10, bytes11, bytes12, bytes13, bytes14, bytes15, bytes16, bytes17, bytes18, bytes19, bytes20, bytes21, bytes22, bytes23, bytes24, bytes25, bytes26, bytes27, bytes28, bytes29, bytes30, bytes31, bytes32, fixed, ufixed, wei, szabo, finney, ether, seconds, minutes, hours, days, weeks, years, public, private, internal, contractor, external, indexed, pure, constant, view, and payable.

8. $\ln(\text{numFunctions}/\text{length})$ - Log of the number of functions divided by file length in characters.

9. $\ln(\text{NumMacros}/\text{FileLength})$ - macros are not supported by Solidity.

10. $\ln(\text{assemblyKeyWords}/\text{FileLength})$ - Log of the number of assembly directives divided by file length in characters: Assembly, break, continue, case, default, return, address, byte, let, switch, function, for, if.
11. *nestingDepth* - Highest degree to which control statements and loops are nested within each other. The structured considered are: while, do-while, if, else, else-if, for, blocks, and statements. The nesting depth was calculated in while, do-while, for, and if structures.
12. *branchingFactor* - Branching factor of the tree formed by converting code blocks of files into nodes.
13. *avgParams* - The average number of parameters among all functions.
14. *avgReturnParameters* - The average number of parameters among all functions returns.
15. *stdDevNumParams* - The standard deviation of parameters among all functions.
16. *stdDevNumReturnParams* - The standard deviation of return parameters among all functions.
17. *avgLineLength* - The average length of each line.
18. *stdDevLineLength* - The standard deviation of the character lengths of each line.

C.1.2 Layout features

1. $\ln(\text{numTabs}/\text{length})$ - Log of the number of tab characters divided by file length in characters.
2. $\ln(\text{numSpaces}/\text{length})$ - Log of the number of space characters divided by file length in characters.

3. $\ln(\text{numEmptyLines} / \text{length})$ - Log of the number of empty lines divided by file length in characters, excluding leading and trailing lines between lines of text.
4. *whiteSpaceRatio* - The ratio between the number of whitespace characters (spaces, tabs, and newlines) and non-whitespace characters.
5. *newLineBeforeOpenBrace* - A boolean representing whether the majority of code-block braces are preceded by a newline character.
6. *tabsLeadLines* - A boolean representing whether the majority of indented lines begin with spaces or tabs.

C.1.3 Syntactic features

1. *MaxDepthASTNode* - Maximum depth of an AST node.
2. *ASTNodeBigramsTF* - Term frequency AST node bigrams.
3. *ASTNodeTypesTF* - Term frequency of all possible AST node type excluding leaves. In our case there were 122 such node types.
4. *ASTNodeTypesTFIDF* - Term frequency inverse document frequency of all possible AST node type excluding leaves.
5. *ASTNodeTypeAvgDep* - Average depth of 58 possible AST node types excluding leaves.
6. *solidityKeywords* - Term frequency of all Solidity keywords.
7. *CodeInASTLeavesTF* - Term frequency of code unigrams in AST leaves.
8. *CodeInASTLeavesTFIDF* - Term frequency inverse document frequency of code unigrams in AST leaves.
9. *CodeInASTLeavesAvgDep* - Average depth of code unigrams in AST leaves.

Vita

Candidate's full name: **Shlomi Linoy**

Universities attended (with dates and degrees obtained):

University of New Brunswick, Expected May 2023, Ph.D.

College of Management Academic Studies, 2009, MBA (High technology industries).

College of Management Academic Studies, 2005, B.Sc. (Computer Science).

Conference Publications:

[C1] **Shlomi Linoy**, Suprio Ray and Natalia Stakhanova, 2022, August. Authenticated Multi-Version Index for Blockchain-based Range Queries on Historical Data. In 2022 IEEE International Conference on Blockchain (Blockchain) (pp. 1-10). IEEE.

[C2] **Shlomi Linoy**, Suprio Ray and Natalia Stakhanova, 2021, December. EtherProv: provenance-aware detection, analysis, and mitigation of Ethereum smart contract security issues. In 2021 IEEE International Conference on Blockchain (Blockchain) (pp. 1-10). IEEE.

[C3] **Shlomi Linoy**, Mahdikhani, H., Suprio Ray, Rongxing Lu, Natalia Stakhanova and Ali Ghorbani, 2019, July. Scalable privacy-preserving query processing over Ethereum blockchain. In 2019 IEEE International Conference on Blockchain (Blockchain) (pp. 398-404). IEEE.

[C4] **Shlomi Linoy**, Natalia Stakhanova and Matyukhina, A., 2019, October. Exploring Ethereum's blockchain anonymity using smart contract code attribution. In 2019 15th International Conference on Network and Service Management (CNSM) (pp. 1-9). IEEE.

Journal Publications:

[J1] **Shlomi Linoy**, Natalia Stakhanova, Suprio Ray, and Eric Scheme, 2022. Authenticated Range Querying of Historical Blockchain Healthcare Data using Authenticated Multi-Version Skip List. *ACM Distributed Ledger Technologies: Research and Practice*. Special Issue on Recent Advances of Blockchain Evolution: Architecture and Performance. (under review)

[J2] **Shlomi Linoy**, Natalia Stakhanova and Suprio Ray, 2021. De-anonymizing Ethereum blockchain smart contracts through code attribution. *International Journal of Network Management*, 31(1), p.e2130.

Workshop Publications:

[W1] **Shlomi Linoy**, Suprio Ray and Natalia Stakhanova, 2020, April. Towards Eidetic Blockchain Systems with Enhanced Provenance. In 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW) (pp. 7-10). IEEE.