University of Zurich^UZH

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Opening Pandora's Box: An Analysis of the Usage of the Data Field in Blockchains

Sebastian Küng
Zürich, Switzerland
Student ID: 15-706-013

Supervisor: Eder J. Scheid, Muriel Franco
Date of Submission: April 1, 2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Der global zunehmende Trend digitale Medien zu posten und zu speichern, hat auch Blockchains geprägt. Ihre Fähigkeit, Daten zensurresistent, redundant und permanent zu speichern, ist jedoch durch Grössenbeschränkungen, Konsensregeln und Transaktionsgebühren begrenzt. Trotz dieser Beschränkungen und gegen weitere Barrieren wie komplexe Methoden und Abraten durch Blockchain-Entwickler haben Nutzer Wege gefunden, Daten in Blockchains einzubetten. Ziel dieser Arbeit ist, sowohl die Menge der Daten als auch die Art der in die Blockchains von Bitcoin, Ethereum und Monero eingebetteten Medien zu analysieren. Ein Vergleich der Ergebnisse zwischen diesen Blockchains kann dann Anhaltspunkte dafür liefern, welche Blockchain sich am besten für die Datenspeicherung eignet und wie zukünftige Blockchain-basierte Speicheransätze den Bedürfnissen ihrer Nutzer gerecht werden können.

Um den theoretischen Hintergrund für die Entwicklung einer Lösung zum Analysieren von Blockchain-Daten zu schaffen, werden Blockchains im Allgemeinen und Bitcoin, Monero und Ethereum konkret zunächst konzeptuell eingeführt. Bestehende Methoden zum Einbetten, Abrufen und Analysieren von Daten werden dann in verwandten Arbeiten untersucht und ihre Ziele sowie Resultate verglichen. Dabei wurde das Fehlen von Blockchain-übergreifenden Lösungen sowie von quantitativen Analysen von Blockchain-Inhalten festgestellt. Die Beschreibung der eingesetzten Methoden und Werkzeuge wird schliesslich genutzt, um eine Softwarelösung für diese Arbeit zu entwickeln.

Als Softwarelösung wurde das Blockchain-Parser-Tool zur Identifizierung von eingebetteten Medien in Blockchains entwickelt. Es analysiert direkt die Blockchain-Datenbank und schreibt Ergebnisse in eine eigene Datenbank mit einem generischen Schema. Von dort aus kann der Benutzer das Tool anweisen, die Daten entweder nach Dateitypen oder nach Zeichenketten mit Hilfe mehrerer Tools zu analysieren. Der Benutzer kann dann die Analyseergebnisse entweder mit dem Blockchain-Parser oder über eine Abfrage an seine SQL-Datenbank anzeigen. Der Benutzer kann Blockchains und ihre Datenverzeichnisse dynamisch auswählen.

Der Blockchain-Parser erstellt ASCII-Text- und Dateitypstatistiken. Für jede der untersuchten Blockchains wurden Textdaten und für Bitcoin und Ethereum einige der eingebetteten Mediendateien identifiziert. Einige der erkannten Zeichenketten und Dateien konnten erfolgreich aus der Blockchain-Parser-Datenbank extrahiert werden und werden in dieser Arbeit vorgestellt. Die Anzahl der erwarteten Datentypidentifikationen wurde mit den tatsächlich identifizierten Dateitypen verglichen. Dies lieferte Beweise dafür, dass viele der erkannten Dateien, vor allem diejenigen, die durch kurze, 2- oder 3-Byte lange, “magische Zahlen” identifiziert wurden, in Wirklichkeit falsch positiv waren. Aus Zeitgründen wurde keine automatisierte Dateiextraktion durchgeführt, obwohl einige Dateien als Beweis der Methodik manuell extrahiert wurden.

Zusammenfassend lässt sich sagen, dass die Ethereum-Blockchain die meisten eingebetteten Text- und Mediendaten enthielt, obwohl alle Blockchains tatsächlich zum Einbetten allgemeiner Medien verwendet wurden. Dies bestätigt die Behauptung, dass die Blockchain-Speicherung für die Nutzer verlockend ist, obwohl sie für diesen Zweck weder konzipiert noch besonders effizient ist. Die Blockchain-Speicherung scheint besonders für kleine Bilder und Texte attraktiv zu sein. Ein System, das die Blockchain-Datenspeicherung nachahmen und ersetzen will, sollte wahrscheinlich als öffentliche, nicht zensierbare, dauerhafte, aber grössenbegrenzte globale Pinnwand mit einmaligen Posting-Kosten konzipiert werden.

Von den drei untersuchten Blockchains scheint Ethereum am besten für die Einbettung allgemeiner Daten geeignet zu sein, da das Datenfeld der Transaktionen leicht zu manipulieren ist, Platz für aufeinanderfolgende Daten zur Verfügung steht und das Netzwerk dabei am wenigsten beeinträchtigt wird. Ethereum komprimiert und archiviert die Dateninhalte von EOA-zu-EOA-Transaktionen und speichert diese Daten im Gegensatz zu einigen der Dateneinbettungsmethoden von Bitcoin und Monero nicht im Cache oder in speicherintensiveren Datenbanken. Obwohl Monero ebenfalls grosse Mengen an aufeinanderfolgenden Daten zulässt, ist es aufgrund ihrer nachteiligen Auswirkungen auf die Einheitlichkeit der Transaktionen und damit auf die Privatsphäre seiner Nutzer wahrscheinlich weniger für die Datenspeicherung geeignet.

In dieser Arbeit wurden einige Aspekte der Einbettung von Daten über verschiedene Blockchains hinweg ausgelassen, die in zukünftigen Arbeiten weiterverfolgt werden können. Es wurde weder eine vollständige Studie über die Erfahrungen der Nutzer mit der Einbettung von Daten in die einzelnen Blockchains durchgeführt, noch über die konkreten Kosten, die sowohl für den Nutzer in Form von Transaktionsgebühren als auch für das Netzwerk in Form von Verarbeitungs- und Speicherkosten anfallen. Ausserdem könnten weitere Blockchains untersucht werden, da das entwickelte System blockchain-agnostisch ist. In dieser Arbeit wurde jeweils ein Beispiel für das UTXO-, das TXO- und das kontobasierte Transaktionsmodell verglichen. Weitere Blockchains, die eines dieser Transaktionsmodelle implementieren, könnten für Vergleiche analysiert werden. Die Fähigkeiten des Blockchain-Parsers könnten noch verbessert werden. Vor allem sollte er die Fähigkeit erhalten, Dateien aus den unterstützten Blockchains zu extrahieren und zu speichern. Sowohl die Geschwindigkeit als auch die Präzision der Analyse sind Bereiche, die Gegenstand zukünftiger Untersuchungen und Entwicklungen sein können.

Abstract

Since the inception of the Bitcoin blockchain in 2009 with the inclusion of the message "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks" in its genesis block, blockchains have been used to store generic media. These include text, images, and documents. However such media is often not easily discoverable in the blockchains and is embedded deep within their binary data structures. The main goal of this thesis is to design and implement a tool that scans blockchains for their media content. The software tool developed for this work, the blockchain-parser, is capable of detecting text strings and files embedded in blockchains. The blockchains of the Bitcoin, Monero, and Ethereum cryptocurrencies were examined to find commonalities and differences between different blockchains in terms of their generic media storage usage. Prior related work has focused on the methods for storing media in Bitcoin. This thesis provides statistics and examples of the blockchain-parser's detected media across Bitcoin, Monero, and Ethereum, which are presented and discussed herein. It concludes that Ethereum has been the most-used blockchain for media data storage of the three and might also be the best-suited blockchain for this task.

Seit Anbeginn der Bitcoin-Blockchain im Jahr 2009 und ihrer in ihrem Genesis-Block enthaltener Nachricht "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks", wurden Blockchains verwendet um Medien, wie zum Beispiel Texte, Bilder und Dokumente, zu speichern. Allerdings sind solche Medien in den Blockchains oft nicht leicht auffindbar und tief in ihren binären Datenstrukturen eingebettet. Das Hauptziel dieser Arbeit ist es, ein Werkzeug zu entwerfen und zu implementieren, welches Blockchains nach ihren Medieninhalten durchsucht. Das für diese Arbeit entwickelte Werkzeug, der Blockchain-Parser, ist in der Lage in Blockchains eingebettete Textzeichenfolgen und Dateien zu erkennen. Die Blockchains der Kryptowährungen Bitcoin, Ethereum und Monero wurden untersucht um Gemeinsamkeiten und Unterschiede zwischen verschiedenen Blockchains hinsichtlich ihrer eingebetteten Medien zu erkennen. Frühere verwandte Arbeiten konzentrierten sich auf die Methoden zum Speichern von Medien in Bitcoin. Diese vorliegende Bachelorarbeit präsentiert Statistiken und Beispiele der mit dem Blockchain-Parser erkannten Medien in Bitcoin, Monero und Ethereum und diskutiert jene. Es wird gefolgert, dass Ethereum von den dreien, die am häufigsten verwendete Blockchain für die Speicherung von Mediendaten ist und auch die am besten geeignete Blockchain für diese Aufgabe sein könnte.

Acknowledgments

I would like to thank and express my gratitude to my supervisor Eder John Scheid for his continued editorial support during the entire duration of the writing of this thesis and for providing the account-based and UTXO-based transaction model figures. I would also like to thank him for the trust placed in me by letting me work on this interesting topic that was originally scoped for a master thesis.

I would also like to thank the CSG group for providing me with a VPS equipped with a generous amount of HDD storage for the Ethereum data analysis.

Contents

Chapter 1

Introduction

Blockchains offer unique data storage capabilities: Availability everywhere with an internet connection, data redundancy by replication across thousands of devices, and retention guaranteed for a very long time. Since the inception of *Bitcoin*, arguably the first "blockchain", they have not only been used to store transactions recording the transfer of value but also to permanently store generic media including text, images and documents [2]. Embedding generic media in blockchains is typically not done through methods designed for this purpose, but rather leverages transaction data fields that are usually used to encode additional transaction information or smart contracts. The demand for a blockchain data store gave rise to services offering simplified storage of generic media in Bitcoin such as eternitywall [22] and apertus [21], launched in 2015 and 2013 respectively.

Using blockchains to store generic media instead of financial data is controversial among developers of blockchains. Recording this data eats into resources that could otherwise be used to verify and store transactions recording value transfers. In a Bitcoin mailing list entry [44], Bitcoin developer Greg Maxwell pointedly describes this duality between storage and utility as follows:

> Since Bitcoin is an electronic cash, it *isn't* a generic database; the demand for cheap highly-replicated perpetual storage is unbounded, and Bitcoin cannot and will not satisfy that demand for non-ecash (non-Bitcoin) usage, and there is no shame in that.

Blockchain storage capabilities are limited both by restrictions on transaction size and economically with the help of transaction fees. Alternative systems combining the capabilities of blockchain and other distributed data storage approaches, like storj [31] and filecoin [40], seek to provide the features of storing media in blockchains like Bitcoin, at a much lower cost. Such systems are not further discussed in this work but are mentioned here for completeness.

In this thesis, an empirical study is conducted on some of the existing media stored in the Bitcoin, Ethereum, and Monero blockchains. To this end, existing work on generic blockchain media storage is presented and compared. Informed by this existing work, a new approach towards detecting and analyzing various media in the aforementioned three blockchains is designed and implemented as a command-line tool, the blockchain-parser [37]. With its help, statistics of generic media in blockchains are compiled and evaluated.

1.1 Motivation

While other works focused on either exposing different methods for data storage in a blockchain or its possible detrimental effects on a blockchain, this work seeks to provide an overview of generic media data storage in multiple blockchains. It is desirable to understand the scale at which blockchains are used for data storage and for which type of data users find this storage alluring. Next to informing blockchain researchers and developers, this can also act as insight for other services that seek to copy some of the capabilities provided by blockchains. Further, this work presents differences between blockchains in terms of their generic data storage capabilities and usage. Although this thesis does not argue in favor of storing generic media in a blockchain, a reader can use the information presented here to select a suitable blockchain for posting media in public. A brief description of how each blockchain stores its data is provided not only to describe how data can be read from the blockchain but also to give the reader an idea of which trade-offs have been made by each blockchain to cater to their respective features. The thesis may also be of interest to digital archaeologists who in the future would like to extract messages and files embedded in blockchains.

1.2 Contributions

The thesis introduces a generic blockchain parser and blockchain-embedded media analysis tool. It provides statistical analysis of data embedded into three popular blockchains: Bitcoin, Ethereum, and Monero. These three blockchains were selected since they offer varying capabilities and data embedding methods for the user. For each of the blockchains, the tool implements a parser reading their blockchain data directly from their database files. The parsed data is stored in a separate database with a blockchain-generic schema and further analyzed for its potential media content. The tool is capable of identifying ASCII strings and the file type of embedded media.

The analysis results as well as the deployment setup of the blockchain-parser for each blockchain are visualized. By analyzing the visualized data in histograms and tables, as well as attempting to estimate the expected number of false positive media detections, the results are discussed and evaluated. Each of the examined blockchains contained both text and generic media. It is assessed which of the three blockchains is utilized the most as a data store.

1.3 Thesis Outline

Including this introduction, the work consists of 6 chapters. Chapter 2, the background, serves as a theoretical springboard into the subject matter. It starts by defining blockchains in Section 2.1 alongside their deployment types, use of consensus algorithms, node types, and common transaction models. Section 2.2 describes what data blockchains persist on disk, providing detail on data stored by Bitcoin, Ethereum, and Monero. Approaches taken by prior work when analyzing blockchain data, especially the type of data analyzed and tools used, are summarized and compared in Chapter 3. Chapter 4 provides an outline of the high-level design and implementation details for the blockchain-parser developed in this work. In Section 4.1 the design of the blockchain-parser including its components as well as the database schema are presented. In the following Section 4.2 the implementation of the blockchain-parser is discussed by explaining how the various blockchains are parsed, which external libraries are used, and defining the media detection algorithms. It also provides usage examples and how analysis results can be viewed. In Chapter 5 the results of running the blockchain-parser on Bitcoin, Ethereum, and Monero are evaluated by visualizing the detected contents and discussing patterns arising between the blockchains. Finally, Chapter 6 gives a summary of the thesis, presents conclusions from the evaluation, and gives an outlook on potential future work.

Chapter 2

Background

This chapter outlines the main concepts involved in this thesis. Section 2.1 defines blockchain, Section 2.2 describes the features and transaction structure of Bitcoin, Monero, and Ethereum, as well as how each of them may be used to embed generic media.

2.1 Blockchain Definition

From its original meaning describing the data set of the Bitcoin transaction graph [48], blockchain has become an umbrella term for distributed database protocols implementing a key idea from the original Bitcoin whitepaper. The whitepaper describes a timestamp server that orders ownership transfer of an electronic coin into a chain of blocked together transactions [48]. A Bitcoin transaction typically consists of one or more inputs, containing the signature, and one or more outputs, determining what the next signature has to verify against. In the simplified transaction model of the Bitcoin whitepaper, a transaction input contains a signature that can be externally verified against the previous transaction's output public key in the transaction chain, thus authenticating coin ownership transfer. However, blockchains, in general, may have different transaction mechanisms and formats, transferring assets, information, or triggering events like further transactions [7]. While Bitcoin nodes operate without prior permission, blockchains may also be deployed in permissioned settings [7].

2.1.1 Double Spending Problem

Bitcoin practically solved the problem of spending a digital coin twice (*i.e.,* the double-spending problem) [48]. If a verifier of transactions detects transactions spending the same coin twice, they need to know which transaction happened first, to reach the same conclusion as another verifier. To this end, a timestamp server creates a new timestamp by grouping transactions into a block, appending the current time, and hashing it together with the previous timestamp. Thus, the timestamp server establishes the order of transactions. Verifiers can rely on the timestamp server to provide a canonical transaction order, while they only need to check signature validity and that the timestamp server does not produce blocks containing transactions twice [48]. Hence, a general blockchain data structure is a timestamped, ordered, back-linked list of blocks of transactions [2].

2.1.2 Deployment Types

Beyond timestamped blocks of transactions, the literature extends the definition of blockchain to its network topology [20]. Blockchains replicate their state and historical data among many nodes in a distributed network [7]. Further, such a replication might present different access restrictions. Permissionless blockchains allow anybody with an internet connection to participate as a full member in the distributed network, "as well as write and read transactions" [7]. Permissioned blockchains restrict the data availability, the possibility to write transactions, or the verification of transaction validity, by either running the blockchain on a non-public network or rules in the consensus algorithm [7].

2.1.3 Consensus Algorithm

Blockchains have consensus algorithms defining the execution of the timestamp server as a multi-party computation between nodes in the distributed network. A node, more commonly referred to as a miner or block producer in this context, provides a proof to authenticate their turn in the multi-party computation. In Nakamoto's Proof of Work (PoW), a nonce in the block is incremented by a miner until a value is found that gives the block's hash the required number of zero bits [48]. A miner solving the proof of work for a block is called the leader for that particular block [7, 68]. Blockchains may employ a host of different consensus algorithms [7, 68]. Next to PoW, Proof of Stake (PoS) selects the leader based on his stake in the underlying coin [7], while Proof-of-Authority (PoA) selects from a pre-determined set of possible block producers [23]. There also exist many hybrid forms of these algorithms, including classical Byzantine Fault Tolerant algorithms [7].

2.1.4 Node Types

Nodes in the distributed network of a blockchain may have different roles. Lightweight nodes only collect a subset of transactions, for example, transactions involving a specific public key [11, 61]. Full nodes validate all transactions and blocks set out by the rules of the blockchain. The common enforcement of rules by full nodes gives rise to consensus [11]. Archival nodes retain the entire history of transactions and blocks. Additionally, archival nodes are also capable of serving the complete transaction and block history to other nodes [61]. This work analyzes some of the transaction and block data that archival nodes retain.

2.1.5 Transaction Models

The blockchains analyzed here are broadly distinguished between blockchains with an input-output and an account-balance transaction model. Input-output based transaction models may be further divided into Transaction Output based (TXO-based), where a transaction input may reference outputs from any previous transaction [36], and Unspent Transaction Output based (UTXO-based), where a transaction input may reference only previously unspent transaction outputs [48, 2, 7]. In UTXO-based blockchains, the current system state is described by the set of UTXOs [7]. Input-output based transactions are akin to analog cash-and-change transactions [7]. Figure 2.1 depicts an example of a UTXO transaction. To spend an amount x to a recipient, a transaction originator has to assemble unspent outputs with a total summed value of $y >= x$. The originator then creates a new transaction output with an amount x addressed to the recipient and a *change* output $y - x$ addressed back to the originator [48, 7, 2].

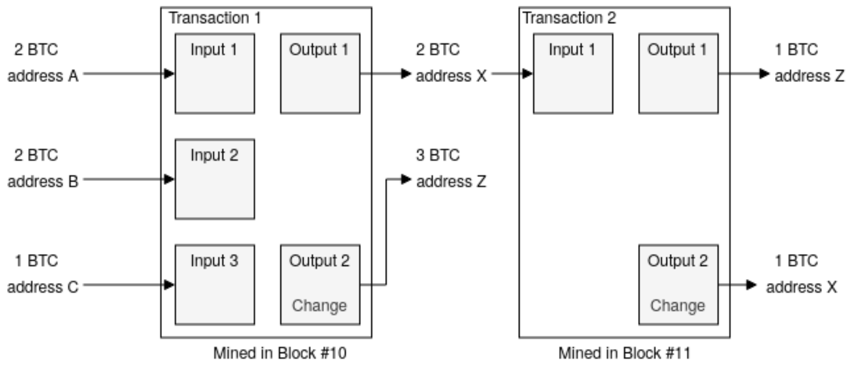


Figure 2.1: Input-output transaction model

Account-balance model transactions transfer amounts by directly subtracting from originator and adding to recipient accounts [7, 72]. Figure 2.2 is an example of an account-balance transaction. The blockchain state keeps track of all accounts and their associated balances.

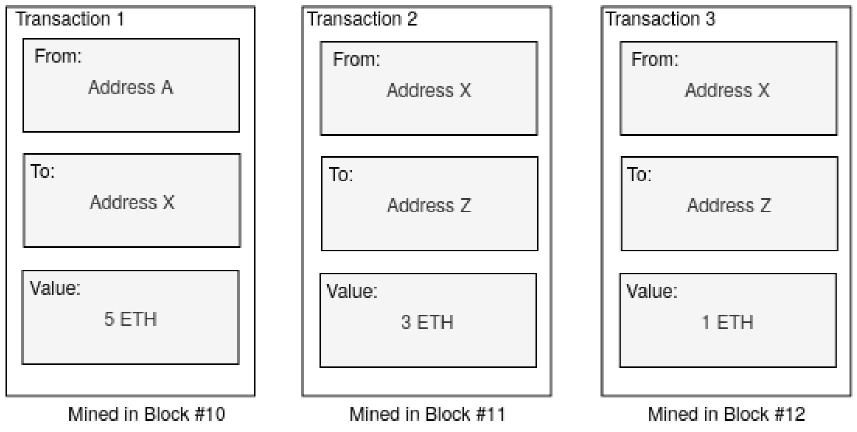


Figure 2.2: Account-balance transaction model

2.2 Blockchain Data

Transaction and block data in a blockchain need not strictly be information describing the ownership transfer of coins and their timestamps [2]. Next to operations allowing for complex transfer constructions and contracts, a user may write arbitrary data into a blockchain [43]. Some blockchains allow writing arbitrary data by design, making specific transaction and block fields and procedures available for the user. Other methods may also be leveraged to embed data or convey additional meaning: Values like public key hashes can be generated until they reach a desired value, scripting languages may allow enough flexibility to embed additional data, and fields that are only used conditionally, *e.g.,* Bitcoin's locktime, allow storage of additional small data. Depending on the transaction model, different data storage types are used across blockchains.

Depending on the transaction model blockchains also have different performance requirements for the storage database. Monero, operating under the TXO-model, requires quick lookup of spend proofs across the entire blockchain, necessitating a fast key-value store: The Lightning Memory-Mapped Database (LMDB) [17]. Bitcoin, operating under the UTXO-model, only needs quick key-value lookups for its UTXO set, which is stored in a LevelDB database [32]. Block data is not queried as intensively but is still retained for cases of chain reorganizations and serving blocks to the peer-to-peer network. The complete block bodies are stored in flat binary files and use a dedicated LevelDB database as a lookup index.

Ethereum, or rather the go-ethereum implementation [71], stores data in multiple LevelDB tables. These include recent blocks and transactions as well as the account state data structures. Older blocks are not persisted in the database, since their quick availability is not needed to maintain and update the account states. Instead, they are compressed with snappy [28] and written to flat binary files. An index to the block's location within these files is loaded into memory.

2.2.1 Bitcoin

Bitcoin uses the UTXO transaction model [48], where outputs contain scripts in a simple, Forth-like Turing-incomplete scripting language, that may be unlocked by corresponding scripts in inputs [2]. When transactions are validated, an input's script is evaluated together with the script of the output being spent [2]. Transactions valid by Bitcoin's consensus rules may additionally be either standard or non-standard. Standard transactions abide by a set of rules enforcing among other criteria which script `OP_CODES` may be used, what templates they should adhere to, and with what data the transaction fields should be populated. Non-standard transactions are not relayed on the Bitcoin distributed network [2]. Standardness checks may be subverted by miners including non-standard transactions in blocks. Additionally, the Pay-to-Script-Hash (P2SH), and Pay-to-Witness-Script-Hash (P2WSH) output script templates allow script `OP_CODES` in a corresponding input `scriptsig` respectively `witness` that would otherwise be non-standard in output scripts. This flexibility allows the embedding of arbitrary data in an input `scriptsig` spending a P2SH output and more generally in non-standard transaction scripts [43].

Other standard transaction templates, such as Pay-to-Pubkey (P2PK), Pay-to-Multisig (P2M), Pay-to-Pubkey-Hash (P2PKH), and Pay-to-Witness-Pubkey-Hash (P2WPKH) may be exploited to hold arbitrary data. The respective public key and public key hash values may be replaced with arbitrary data in the transaction output. Next to the transaction fee, these transactions have additional costs for the user. Their outputs are unspendable because the user replaces valid public keys and public key hashes with arbitrary data (invalid public keys and public key hashes) [43]. These methods moreover incur costs for the Bitcoin network as a whole, since these unspendable outputs cannot be detected and therefore not pruned, or omitted from caches and the Unspent Transaction Output (UTXO) set. According to [43] embedding data into unspendable, but standard, transactions, is the most economical method for a user and allows storage of up to 92'625 Bytes in the P2MS case across many outputs in a single transaction.

Miners may embed arbitrary data in a coinbase transaction. Because Bitcoin coinbase transactions do not reference any previous transactions, their `scriptsig` has no semantic meaning. Satoshi Nakamoto's message in the first Bitcoin coinbase transaction "*The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*" [2], was embedded in the `scriptsig`. Since the activation of Bitcoin improvement proposal 34 (BIP34), the block height of the coinbase transaction has to be the first item in the `scriptsig` [1], while the rest of the field remains at the discretion of the miner. In practice, BIP34 limits the amount of arbitrary data in a coinbase `scriptsig` to 96 bytes [43].

In August 2017 Bitcoin deployed a backward-compatible upgrade to its transaction structure: Segregated Witness (segwit) [2]. SegWit introduced a new standard transaction output type and moved the signature and input script `OP_CODES` to a new *witness* field. Depending on the serialization this witness field is either represented as a vector alongside the inputs or another field within a transaction input. Segwit transaction outputs, also referred to as witness programms, are versioned [2]. Version "0x00" includes P2WPKH and P2WSH outputs, while version "0x01" includes pay-to-taproot (P2TR) outputs.

Bitcoin script provides the `OP_RETURN` opcode, which allows users to write up to 83 bytes of arbitrary data into a standard transaction output [5]. The Bitcoin developers recommend using `OP_RETURN` to store arbitrary data in a transaction without side effects[2]. `OP_RETURN` outputs are unspendable and can therefore be pruned, omitted from caches, and are not part of the UTXO set. More than one `OP_RETURN` output per transaction is considered non-standard. Various protocols external to Bitcoin leverage `OP_RETURN` to encode additional asset data [5].

Vanity addresses encode meaningful substrings in Bitcoin addresses [4, 25]. P2PKH addresses are Base58-Check encoded hashes of Elliptic Curve Digital Signature Algorithm (ECDSA) public keys [48]. While addresses are not directly stored in the Bitcoin blockchain, user-facing applications like wallets and block explorers can re-construct addresses from transaction input and output data [2]. Vanity addresses may be created by brute-force searching ECDSA public keys until a corresponding address contains the desired substring [4, 25]. An example of this would be "1BoatSLRHtKNngkdXEeobR76b53LETtpyT", where the string "Boat" is contained in the address. Alternatively, the entire string is set to a desired value. Such addresses are used for "Proof-of-Burn", where coins are sent to unspendable addresses, since the corresponding pri-

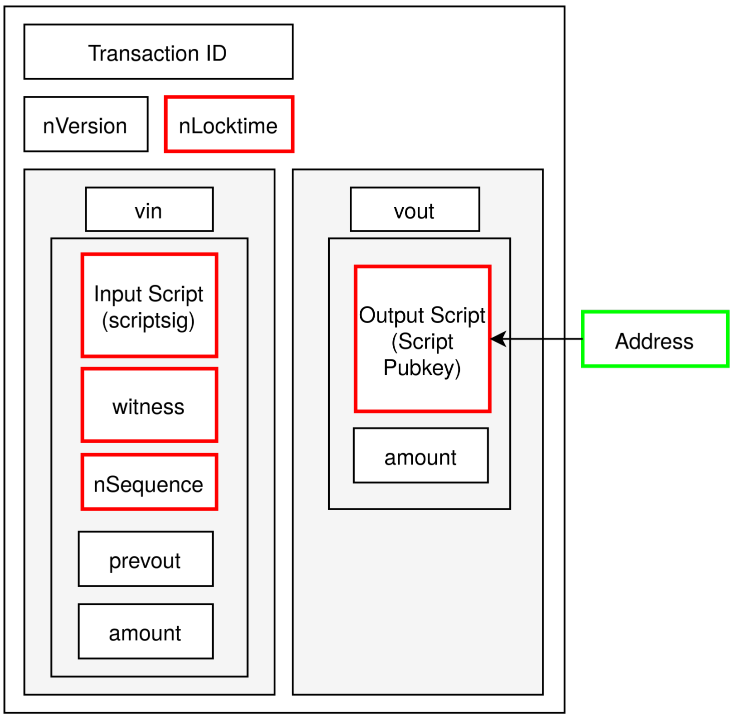


Figure 2.3: Bitcoin transaction fields

vate key is unlikely to be known by anybody [4]. A well-known address is "1CounterpartyXXXXXXXXXXXXXXXUWLpVr", which is used by the Counterparty protocol "Proof-of-Burn" [53]. Data may be distributed over many vanity addresses, contained for example in the inputs and outputs of a transaction [4].

Figure 2.3 shows a possible representation of the fields of a Bitcoin transaction. Marked in red are those fields that have been previously identified to hold arbitrary data embedded by users. The `witness` field is only present for segwit transaction inputs. Alternatively the `witness` field can be serialized and represented outside of the shown `vin` structure. Additionally, the address is marked in green since it is not strictly part of a transaction, but nevertheless conveys information as transported by an output script.

2.2.2 Ethereum

An Ethereum transaction is an authenticated instruction to the Ethereum Virtual Machine (EVM) [72] using an account-balance transaction model. A transaction changes the state of its originator and recipient accounts [3]. While the EVM is capable of Turing-complete computation, the halting problem is sidestepped by the introduction of fees [72]. Fees per specific operations on the EVM are given in gas. Executing transactions drains gas [72] from the originator account. Figure 2.4 shows the fields of an Ethereum transaction. Each transaction has to specify a `gas limit`, the maximum amount of gas the originator is willing to pay. If a transaction costs more gas than the `gas limit`, its execution is halted and the state is rolled back [3]. A transaction also contains a `gas price`, or how many Ethereum base units (Wei) the user is willing to pay per unit of gas [72].

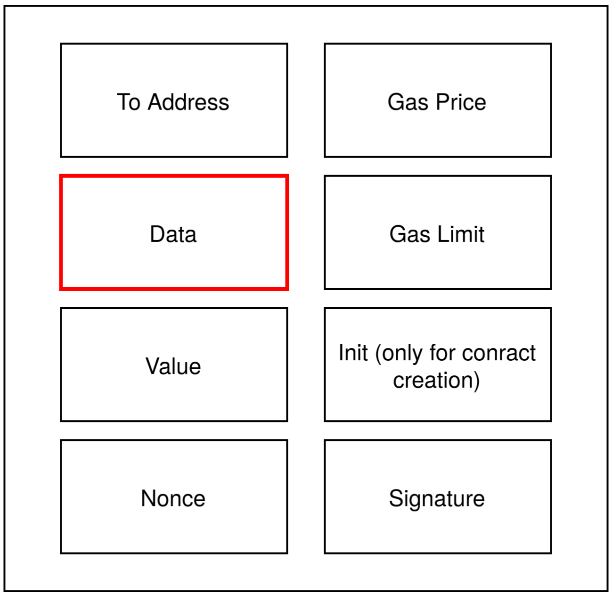


Figure 2.4: Ethereum transaction fields

Ethereum transactions have an input data field (marked red in Figure 2.4), a field without size limits, though bounded in practice by the block gas limit, the maximum amount of gas allowed per block [72]. Ethereum has two types of accounts, Externally Owned Accounts (EOA) and contracts. An Ethereum transaction transfers a message from an EOA to either an EOA or a contract. Dependent on the recipient account type, the data field is interpreted differently. The data field of transactions between EOA's is ignored by Ethereum's consensus rules and may have arbitrary content [3]. The data field in transactions from EOA's to contracts is interpreted by the EVM as a contract invocation. Contract invocations typically contain data selecting the correct function of the invoked contract with its associated function arguments [3]. An EOA may create a new contract by sending a transaction to the reserved "zero" address and including the contract's compiled bytecode in the data [3]. Any non-zero transaction data costs 16 gas per byte [72].

An Ethereum smart contract as contained in the transaction data of a contract deployment transaction is usually compiled from a higher-level language such as Solidity [3]. A Solidity compiler however is not only able to compile the contract code to EVM bytecode, but also produces an application binary interface (ABI) in plaintext JSON that describes the function and data structure interfaces of the contract. The ABI thus defines how later invocations on the smart contract need to be structured. The first 4 bytes of an ABI function's interface SHA3-hash are referred to as its *methodId* [3]. A transaction invoking a smart contract's function includes the *methodId* as well as the function arguments in the data field [3].

2.2.3 Monero

Monero is based on *cryptonote*, employing one-time keys to hide the recipient identity and ring signatures to hide the originator within a group of potential signatories [66, 36]. The keys in the ring signature are sampled from prior transactions. For a ring signature with size n, a signers own key pair as well as $n-1$ *decoy*, or alternatively called *mixin*, keys are selected. Amounts are additionally encrypted with *confidential transactions* [36, 49]. While Monero transactions have inputs and outputs, transactions may refer to any previous outputs and thus follow the TXO-based transaction model [36]. A key image of the signer's true private one-time key is added to every transaction input, ensuring double-spend protection by uniquely tagging an input signed by a specific private key [36].

Monero transactions and blocks have a `tx_extra` field. The contents of `tx_extra` are disregarded by consensus [36]. `tx_extra` may be populated with arbitrary data by users. However, transaction- and block-relevant information is also stored in `tx_extra` and identified with reserved one byte tags [36]. These include information for wallets, such as transaction public keys, encrypted payment IDs, and extra nonces for miners [36]. Using `tx_extra` for arbitrary information storage is controversial, since adding plaintext information may de-anonymize single users and decreases transaction uniformity, which may be detrimental to the privacy of Monero as a whole [60].

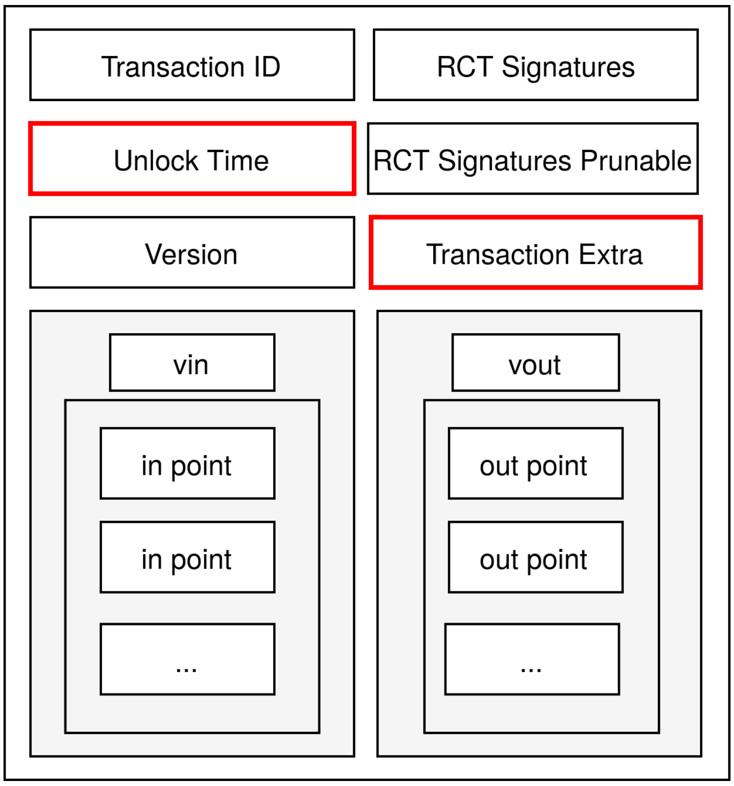


Figure 2.5: Monero transaction fields

Figure 2.5 show the structure of a Monero transaction with fields that may be leveraged by users to store arbitrary data marked red. Similar to Bitcoin's locktime, Monero's unlock time may also be used to embed small amounts of data.

The contents of `tx_extra` can be serialized and deserialized with the help of tags and encoded field lengths. For a normal Monero transaction, `tx_extra` contains a single transaction public key and payment ID. A payment ID is used by the recipient to distinguish outputs received. It may either be an 8-byte encrypted payment

ID, or a 32-byte long payment ID that contains pre-communicated data for identification [36]. Since monero consensus version 10 every transaction created by the Monero reference wallet contains a single transaction public key and encrypted payment ID. The Monero code in src/cryptonote_basic/tx_extra.h defines the commonly used content tags used for serializing `tx_extra` content. The following decoded representation thus matches the below hexadecimal `tx_extra` field of the transaction 9b6318cbd3af1b4ee7f17336e1a7124aaaef4a9fcc43d1160255e3a30ee39ac6:

```
TX_EXTRA_TAG_PUBKEY <pubkey> TX_EXTRA_NONCE <9> TX_EXTRA_NONCE_PAYMENT_ID
<encrypted_payment_id>
```

```
01 540aa530a9bfeae381b0993287d28627f3584f38a6bf047012d8e134a5c03ca0 02 09
01 7b5175ad4f853fc7
```

2.3 Steganography and Steganalysis

Steganography describes a class of algorithms allowing covert communication through an otherwise open communication channel without an active adversary noticing that covert messages are exchanged. An otherwise inconspicuous message, or *cover object* is transformed into a *stego object* containing a covert message. To an outside observer the difference between *cover object* and *stego object* should be undetectable [16]. As a further constraint on a steganographic algorithm Kerckhoffs principle of open cryptographic design should ideally hold [13]. Steganography is often done in conjunction with media like digital images, audio or movies. Otherwise inconspicuous steganographically prepared media may be publicly posted, while in actuality containing a covert message. Modern steganographic algorithms employ public key cryptography to generate a shared *stego key* used to encrypt and embed the covert message. Recipients then generate the same shared key and use it to extract and decrypt the covert data [16, 18].

Steganalysis encompasses methods to detect *stego objects* and potentially extract the covert message [16]. Steganalysis typically targets weak points of steganographic algorithms. One of the simpler steganographic algorithms is Least Significant Bit (LSB) replacement, where every least significant bit in the cover medium is replaced by a bit of the covert message, thus only slightly altering the *cover object.* While potentially convincing for a casual viewer of the *stego object*, LSB replacement is usually easily detectable in the *stego object* by analysis of byte asymmetry [34].

Openpuff is among a host of popular steganography tools. Its main purpose is steganographically hiding messages in various media, such as images and videos. Potential usage of Openpuff in content found on blockchains could be specifically analyzed since the Bitcoin Wiki's mention of it might hint at potential usage in Bitcoin and other blockchains [6]. OpenPuff is known to use LSB replacement for its image steganography, which lends itself to statistical attacks [12].

Chapter 3

Related Work

This chapter describes the related approaches to the work presented in this thesis in Section 3.1 and compares them in different dimensions in Section 3.2. Its purpose is to collect prior experiences relating to analyzing blockchain content, from which the development of the blockchain-parser tool should be informed.

3.1 Approaches

Analyzing and discussing the media content of blockchains has been a point of interest for cryptocurrency enthusiasts, hobbyists, and reporters alike since the Bitcoin genesis block with its timestamp message was mined [24, 67, 26]. Hence, academic examinations have been undertaken by several works. In the following, related work is presented for the Bitcoin, Ethereum, and Monero blockchain.

[10] presents an analysis of the prevalence and types of non-standard Bitcoin transactions. Their presented transaction data was collected with the help of a Bitcoin Core node and stored in a PostgreSQL database. In [9] one of the authors of [10] provides additional insights into the used visualization workflow ("BlockChainVis"), detailing that all block or transaction data retrieved from the Bitcoin API enters a single PostgreSQL database from which it may later be queried. [9] also stresses that RAM usage is a metric when selecting a database for querying many, maybe related, transactions from a blockchain. [10] identifies that non-standard transactions make up the small fraction of 0.02% of all transactions at block height 550'000. Though not stated explicitly how this number was computed, likely, the standard transaction output types (P2PKH, P2PK, OP_RETURN, P2SH, P2WPKH, P2WSH) were counted first. By exclusion, the remaining are non-standard transactions. A detector for the standard transaction output types can be constructed by analysing the used `OP_CODES` in the output script pubkey. While [10] provides histograms of common patterns of non-standard transactions, these fail to include any spent transactions making it presently unclear what their actual usage across the history of Bitcoin is.

UTXOs are more generally examined by [19] by analyzing and quantifying the Bitcoin UTXO set. They present their own tool (STATUS [57]) for reading and analyzing the

Bitcoin LevelDB database files in which the Bitcoin UTXO set is stored. The UTXO set parser implemented for this thesis borrows from their work. Their analysis shows the evolution of the amount of output types as a function of their block height showing stagnation of non-standard output and a continuous increase of standard output types at block height 491868. Further, the UTXO-set is made up of 82% P2PKH outputs, 17.1% P2SH outputs, 0.1% P2PK outputs with the remaining 0.8% divided into a 99.8% majority of P2MS outputs around 0.1% SegWit outputs and 0.1% non-standard outputs.

UTXOs that are marked unspendable by the `OP_RETURN` `OP_CODE` are studied in [5]. More specifically [5] investigates the data footprint of protocols built on top of data embedded into Bitcoin with `OP_RETURN`. They further distinguish the data and protocol purpose by their share of the total usage of `OP_RETURN` at their time of data collection on the 15th of February 2017. The majority of identified `OP_RETURN` transactions are related to asset protocols (26.7%), followed by notary services (8.3%) and digital arts (5.5%), while protocols whose purpose could not be identified make up 10.8% of the share. A further 15.7% contained no data alongside the `OP_RETURN` `OP_CODE` leaving 32.8% with unknown classification. Their detection method relies on concrete protocol identifiers appended to the actual data. The Omni-Protocol for example prepended the identifier `omni` to their data.

Bitcoin data insertion methods next to `OP_RETURN` are explored in [59], whose authors attempt to classify, analyze and compare data insertion methods in Bitcoin. Their analysis distinguishes the following insertion methods:

1. Coinbase: Data embedded in the input of a coinbase transaction
2. Pay-To-Fake-Key-Hash (P2FKH): Replaces the public key hash of a P2PKH output with data
3. Pay-To-Fake-Key (P2FK): Replaces the public key of a P2PK output with data
4. `OP_RETURN`: Data stored in `OP_RETURN` outputs
5. Pay-To-Fake-Multisig (P2FMS): 1-of-n P2M output retaining at least one real key and replacing the rest with data
6. Pay-to-Fake-Script-Hash (P2FSH): Replaces the script hash of a P2SH transaction with data
7. P2SH with `OP_DROP` redeem script: A P2SH transaction where the redeem script contains data preceded by the `OP_DROP` `OP_CODE`. Allows spending, but is malleable, since the data is not in the redeem script.
8. P2SH with Data Hash: Part of the redeem script commits to and checks the data contained in the complete input script. Spendable and protects against malleability.

[59] further goes on to identify the P2SH with `OP_DROP` data embedding method as the most efficient and economic in terms of both costs spent on transaction fees and available data size per transaction. They provide a count of P2FKH usage by counting all P2PKH

UTXOs where the key hash contains at least 18 consecutive printable ASCII characters. These total 129'410 UTXOs storing 2.59 MB while burning 118.96 BTC as of June 7th, 2017. It should be noted that the fake key method also applies to segwit and taproot outputs, though these did not exist at the time of publishing. Reconstructing non-ASCII data is conceded as more difficult, since data may be spread over many outputs and transactions. Though they mention that metadata, like the format, name, and size is often embedded alongside the actual data, no generic method for extracting the data is given. The appendix of [59] contains code examples detailing how data is parsed from the raw block files, lists some TXIDs of transactions that contain images, as well as showcases a few example transaction constructions with the help of the bitcoinj library [56].

Next to analyzing and quantizing generic media data in Bitcoin, [43] discusses some of the legal impacts of storing this data on the Bitcoin blockchain by raising awareness of unsavory content, like links to child pornography websites. They stipulate that these might make the possession of the blockchain illegal for its users in many jurisdictions. They additionally give counts of different file and media types found, including text, images, HTML, other source code, archives, audio, and PDFs in their table 3.1. No methods for either their insertion type or the data content detectors are provided.

Table 3.1: [43]: Distribution of blockchain file types

File	**Via Service?**		**Overall**	**File**	**Via Service?**		**Overall**
Type	yes	no	**Portion**	**Type**	yes	no	**Portion**
Text	1353	54	87.07%	Archive	4	0	0.25%
Images	144	2	9.03%	Audio	2	0	0.12%
HTML	45	0	2.78%	PDF	2	0	0.12%
Source Code	7	3	0.62%	**Total**	1557	59	100.0%

[27] probes transaction data for potential traces of steganography. Their steganalysis focuses on steganographic approaches specific to blockchains and not on steganalysis of media embedded in blockchains. The method for embedding data in the blockchain outlined by [52] gave the impetus for their research. [52] gives a provably secure method for LSB replacement in Bitcoin addresses. [27] notice that this same method also applies to block nonces, which are similarly randomly distributed.

They identify that the problem is further complicated in the case of addresses since they might be spread over many transactions in non-consecutive blocks. Addresses with a covert message from a common originator are likely able to be clustered together by heuristics. For Bitcoin address clustering they mention multi-input, a single change output with the same type same as the inputs, while other outputs are of a different type, and the change value often being the smallest value [46, 33]. The insight that address clusters arise when a common originator enters data into a blockchain can also be used for our analysis cases where data may be distributed over multiple transactions. They note that clustering requires a lot of memory. This work attempts to lower the memory requirement by pre-filtering transactions.

Next to these address clusters, they also create ordered data sets of block nonces in order of the block height and addresses in order of their index within their containing transaction and transaction order within a block. The sequential data is collected with the aid of the Python Bitcoin Blockchain Parser [14] and Ethereum ETL [45] python libraries. Additionally, the address clustering is achieved with the Bitcoin Blocksci [33] library. Further, the scalpel tool is used to potentially detect files within the data.

They then calculate three different metrics that could indicate the presence of LSB replacement steganography. For each cluster or sequence the Shannon entropy, arithmetic mean, and monobit failures are calculated. The Shannon entropy [41] is a metric for the information-theoretic entropy in a piece of data, where P is the probability of a piece of data x_i appearing:

$$E = -\sum_{i=1}^{n} (P(x_i) \log_2(P(x_i)))$$

The Shannon Entropy is thus a measure of the number of bits needed to store the information in a variable. Monobit tests are a measure for randomness in a bitstring by counting how many more ”1” bits than ”0” bits are contained. By calculating a “p” value of these events, bitstrings can be accurately assessed for their randomness.

Finally, [27] present histograms of the arithmetic mean and LSBytes of bitcoin and Ethereum nonces, but failed to detect evidence for steganography and state whether the data passed the Shannon entropy and monobit failure tests. Similarly with the address data and clustered address data. To detect stenographic activity, the authors analyzed histograms of the least significant bytes of their extracted data. Their analysis did not turn up evidence of steganography used in Bitcoin.

Next to Bitcoin, data analysis approaches have also been published and discussed for Ethereum. [73] extracted data from the Ethereum blockchain and developed a clustering and partitioning framework. While clustering and partitioning transactions is out of scope here, their work offers some insights into data extraction from Ethereum clients. Raw block data of Ethereum clients only store transactions as issued by the user. The exact contract state that serves as the input of the transaction is however not stored. To be able to analyze input states, a trace of the contract execution has to be collected.

For smart contract data specifically, [30] classify smart contracts by analysing the data field extracted from transactions collected from Etherscan and traces collected from the Parity Ethereum client. [8] generates histograms of opcodes after collecting contracts from Etherscan. The literature survey conducted for this thesis has however not turned up broad arbitrary media analysis done on the Ethereum blockchain.

Much of the research published on Monero's blockchain focuses on achieving transaction clustering and linkability. For example, [47] extracts data from the Monero blockchain and attempts transaction clustering by two heuristics to break Monero's privacy. Before 2017 Monero users could specify zero additional *mixins*, effectively affording themselves no privacy improvement. Further, the *mixin* sampling potentially allowed guessing the true input by its spend-age. With these heuristics, the authors managed to guess the true signer for up to 40% of transactions. They also remark that transaction linking heuristics that can be applied to identify the true signer, can also be used to exclude signature ring members as *mixins* in other transactions where the true signer is unknown. As a reaction Monero samples *mixins* from a distribution mimicking true spend behavior and enforces constant ring sizes since the year 2018.

[70] investigates possible transaction linkability due to re-use of payment ID data stored in the `tx_extra` field. If two transactions use the same unencrypted payment ID the true signer is revealed if the second transaction uses the change output of the first. Additionally, they identify the possibility for abuse of plaintext data stored in `tx_extra`, potentially proving detrimental to the user's privacy. Commonalities between the embedded data in multiple transactions, for example, embedded E-Mail addresses belonging to the same domain or the synchronicity of their publishing, can be used to link transactions. More recently, hobbyist researchers published a list of text media embedded in `tx_extra` [38]. However, a broader analysis of embedded media in Monero has not been undertaken yet.

3.2 Comparison

Table 3.2 compares the related works described in Section 3.1. For each approach, the target blockchain, the analysis framework, the data analysed, and the analysis scenario were identified. The findings from this comparison are described below.

Bitcoin's arbitrary data storage is well studied and has been used by a variety of services. Three gaps can be identified in the research presented here: A probe of coinbase transaction unlock time and sequence values, automated extraction from the Bitcoin blockchain of generic media embedded therein, and a steganalysis of media embedded in Bitcoin. Although the research groups present a host of different tools, little re-use of tools is observed between the works. The solution here re-uses some of the tools of [27] and [5], namely the Python Blockchain Parser [14], eth-rlp [15] (a sub-dependency of their Ethereum ETL [45] dependency) and GNU strings. It also uses STATUS [57] from [19] to retrieve data from the Bitcoin UTXO set. Further, similarly to [9], the data is written to a single database instance.

The related work presented here has shown that no analysis of media stored in either Ethereum or Monero was found. In Monero specifically there is no analysis of stored arbitrary data in `tx_extra`, though [70] treats the possibility for abuse. Since the literature discusses no proper framework for data extraction on the Monero blockchain, this work relies on a custom software based on the Python LMDB wrapper library [69] and the Monero Python serialization library [35]. The usage of the Ethereum transaction data field for contract code is well studied, especially for classifying different contracts and clustering transactions, but no treatment of arbitrary data stored within it has been found. Overall there is no comparison of arbitrary data storage between the blockchains.

Table 3.2: Comparison of Related Work

Work	Blockchain	Analysis Framework	Data Analysed	Analysis Scenario
[43]	Bitcoin	Unknown	Transactions	Arbitrary data types, legal impacts
[19]	Bitcoin	Bitcoin Core + STATUS [57] + numpy	UTXO set	UTXO set output type counts
[27]	Bitcoin, Ethereum	Bitcoin: Bitcoin Core + Python Blockchain Parser / BlockSci + Scalpel ; Ethereum: Ethereum-ETL + Scalpel	Addresses, Block nonces	Address clustering and stetaganalys, nonce steganalysis
[10]	Bitcoin	Bitcoin Core + PostgreSql	Transactions	Classification and Statistics of Non-standard Transactions
[5]	Bitcoin	Bitcoin Core + Various Shell Utilities (GNU strings, Binwalk, local-blockchain-parser, binary-grep)	Transactions	OP_RETURN based protocol detection and statistics
[59]	Bitcoin	Bitcoin Core + Custom Bitcoin Block Parser + bitcoinj	Transactions	Survey and comparison of data insertion methods
[73]	Ethereum	XBlock-ETH	Blocks, Transactions	Block and transaction partitioning and clustering framework
[30]	Ethereum	Parity client, Etherscan	Transactions (Smart Contracts)	Smart Contract Classification
[47]	Monero	Monerod RPC, Neo4j	Transactions	Transaction Linkablity
[70]	Monero	Unknown	Transactions	Transaction Linkability, Payment ID reuse

Chapter 4

Design and Implementation

Based on the research on the related work to this thesis, this chapter details the design and implementation of a command-line tool as a software solution for parsing blockchain data and writing this data to a tool-specific database for later data analysis and retrieval.

4.1 Design

The solution was designed with three requirements in mind: (1) Optimize the stored data, (2) be as generic as possible under the changing data structures of the blockchain being analyzed, and (3) flexibility towards the data to be identified. The solution is not optimized for data analysis across multiple transactions; it does not provide tools for transaction clustering. Data embedding methods relying on clusters of transactions are therefore hard to detect with the tooling presented in this thesis.

Figure 4.1 depicts the high-level flow of data and its components. Blockchain transactions are first parsed from each blockchain instance. Once parsed, the data passes through pre-processing criteria, *e.g.,* a check if transactions are non-standard. Only if the data passes these, it is written to a database. An analysis engine component then runs detectors on the data for different types of media and if detected applies data labels to the database. Lastly, there is a view component that presents results to the user or he/she can query the database for results directly.

4.1.1 Components

The **transaction parsing** component operates under the assumption that it has access to the raw blockchain data of the respective blockchain. Typically, blockchain implementations provide Application Programming Interfaces (APIs) through HTTP or ZeroMQ [42] servers to query blockchain data. These APIs incur processing time overhead compared to reading the data directly from disk. Thus, the transaction parser trades shorter processing times for direct access to blockchain data files. The parser can also be described as a generic iterator for blockchain data.

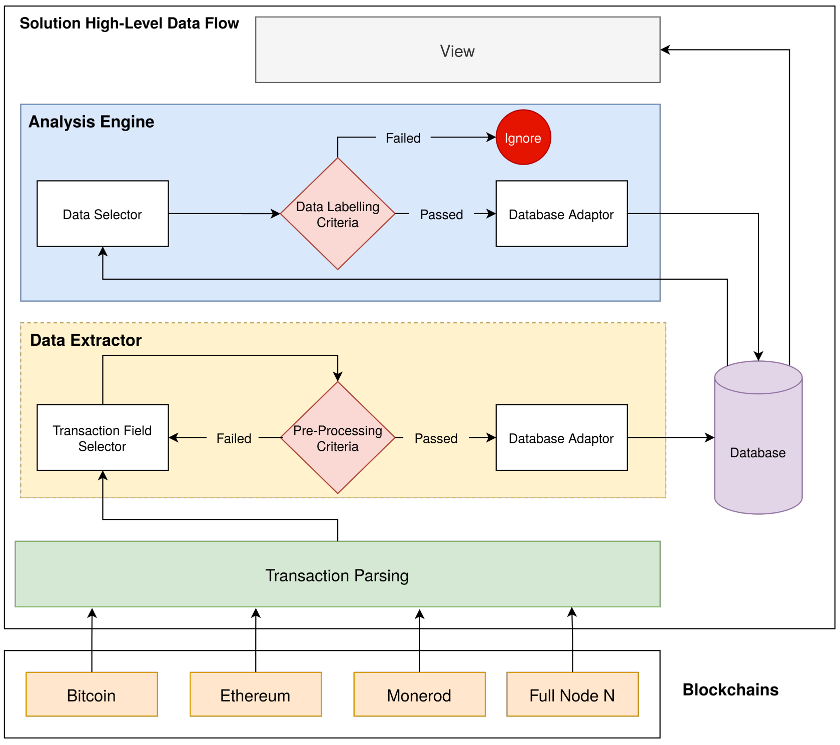


Figure 4.1: Solution High-Level Data Flow

In the **data extractor** component fields of interest from the parsed transactions are selected. The selected fields are then filtered through a set of pre-processing criteria. The selection and filter steps are applied to reduce the amount of data written to the database for further analysis by excluding all transaction fields that are unlikely to contain embedded media. Selecting fields is highly dependent on the underlying blockchain. As described in Chapter 2, the fields suitable for data storage vary between the coins. The pre-filtering criteria may include generally applicable criteria, *e.g.*, checking against a minimum length requirement, but are mostly also specific to the underlying blockchains and attempt to filter out transaction data formats that are unable to hold arbitrary media. Each row of data written to the database must have a unique ID that can be reconstructed from the raw transaction data.

The rows of processed data are then analyzed for potential media content by the **analysis engine** component. Each row is iteratively selected and tested against criteria checking for a specific media type. If a media type is successfully detected, a label for that specific entry is created in the database. Labels are maintained in their own database tables. The data labeling criteria use a combination of blockchain-specific and generic methods. Many blockchain data insertion methods also insert data specific to the operation on the blockchain alongside the actual data. This additional data needs to be stripped both to improve the performance of the generic methods and reduce false positives. Thus, every data labeling criterion first applies blockchain-specific pre-processing before generic data detection.

Finally, the **view** component queries the database to compile results for presentation to the user. To make interaction with the database easier, common adaptors for selecting and writing data are defined. These include the database adaptor and data selector of

Figure 4.1. Compiling results from the labeled data should be done with as many database-intrinsic operations as possible. For example, with the help of SQL queries, instead of processing the data iteratively with a data processing framework, such as Python's NumPy [50]. The database can optimize these queries for both a shorter runtime and reduced memory footprint, as opposed to a data processing framework, in which the user is responsible for the implementation and configuration of such optimizations.

4.1.2 Database

Figure 4.2 depicts the adopted schema of the database. Each row of data is identified by a triple of attributes making it unique. This attribute triple includes the `TXID`, or transaction ID of the transaction the data is contained in, an `EXTRA_INDEX` especially relevant for Bitcoin's case where the exact index of the input or output of a transaction needs to be specified, and a `DATA_TYPE` marking which part of a transaction the data is coming from. This is *Transaction Extra* for Monero, *Data* for Ethereum, and *Input or Output Script* for Bitcoin. Next to this triple constituting the primary key, the actual data in raw bytes, the coin or originating blockchain, and the block height of the transaction are recorded.

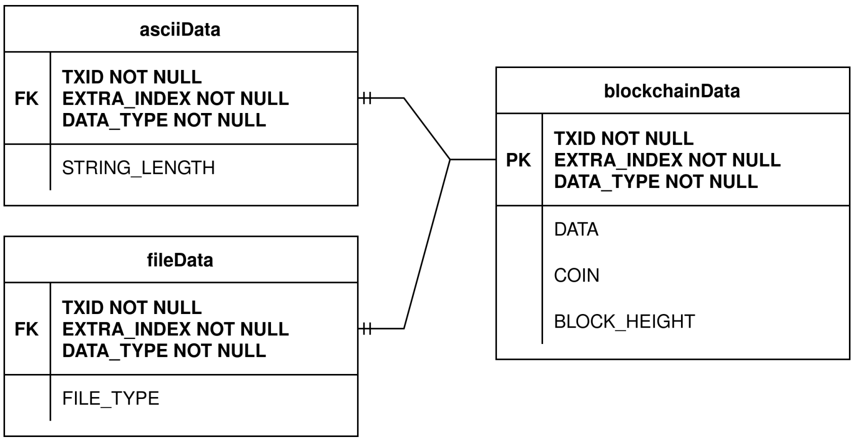


Figure 4.2: Example Database Schema

For this thesis, two detectors are implemented as data labeling criteria in the **analysis engine** component: (i) One for detecting raw ASCII strings and (ii) one for detecting potential files. Each detector iterates over all pre-filtered potential data entries in the `blockchainData` table, runs the data through the detector, and on detection writes the length or type of the data alongside the primary key of the `blockchainData` table as a foreign key to a dedicated table for the detector, meaning either `asciiData` or `fileData`.

4.2 Implementation

This thesis implements the solution's design (*cf.* Figure 4.1) as an open-source command-line tool using Python and a SQLite3 [29] database, called *blockchain-parser*. Its code is available at [37]. Blockchain-parser is a command-line tool operating in three modes: parse, analyze and view. The parse mode implements the **transaction parsing** and **data extractor** components, analyze the **analysis engine** component and view the **view** component. A single, common database ensures data continuity between the different modes. A database adaptor provides common SQL queries for inserting and reading data to define a common schema between blockchains as defined in Figure 4.2. A unique transaction parser and data extractor are developed for each blockchain, though the latter is subclassed under a common super-class interface. The analysis engine is implemented as a monolithic class but contains selected functions that are blockchain-specific to strip unnecessary data. While the view implementation may take a blockchain as an argument to select results, it has no blockchain-specific functionality. Wherever possible the code is annotated with `mypy` types and `docstrings`.

4.2.1 Transaction Parsing Implementation

Table 4.1 shows the tools and libraries used for parsing and de-serializing data from the respective blockchains. A new, custom parser for the *go-ethereum* database format was implemented for this work. Since Monero stores all its data in a single LMDB database, no special data extraction logic was required, beyond the Python-LMDB library [69]. The STATUS [57] tool used to parse the Bitcoin UTXO-set had to be manually ported to Python3. The data serialization could be achieved in each case with additional code required to connect the extraction and serialization libraries. While both `python-bitcoinlib` and `eth-rlp` were straightforward to use, the Monero Python Serialization Library implements its functions only asynchronously. This required the additional use of Python's *asyncio runtime* to drive the execution of these functions. Each transaction parser is implemented as a Python iterator within an instantiation of a data extractor component class.

Table 4.1: Blockchain Parsing Tools

Blockchain	**Data Extraction**	**Data Serialization**
Ethereum	Custom	eth-rlp [15]
Monero	Python-LMDB [69]	Monero Python Serialization Library [35]
Bitcoin	Python Bitcoin Blockchain Parser [14] + STATUS [57]	python-bitcoinlib [62]

Ethereum

The custom implementation of the Ethereum data parser for this work is outlined in the following two paragraphs. The go-ethereum [71] Ethereum node stores block bodies and their headers in separate indexes and files. For this reason, two separate iterators were created for block headers and block bodies. Listing 4.1 shows the implementation of the block body iterator. It provides access to both the LevelDB tables for recent data and the so-called "Freezer" tables, using go-ethereum's terminology, for historical blocks and transactions. All the data in the "Freezer" tables are stored in a series of numbered data files. An index file keeps track of each data blob's start and end positions within the data files with fixed-size entries. Each index entry is a tuple of a 2-byte integer representing the file number and a 4-byte integer representing the offset byte where the next blob of data is expected. Whether the data in the "Freezer" tables need to be decompressed with snappy [28] is additionally identified by the file extension of the data files. The index entries are ordered by block height, facilitating a time-ordered retrieval of data.

By reading the index files for both the block body and header tables, it is thus possible to retrieve data by block height, as done in Listing 4.1 (line 17). If no data is retrieved, the iterator falls back to retrieving from the LevelDB database. Two queries are required to retrieve blocks or headers from the database. The first query uses the block height as a key and retrieves the block hash, the second uses both the block hash and height as a key and retrieves the full block. In both the "Freezer" tables and the LevelDB database the data is encoded using Ethereum's Recursive Length Prefix (RLP) encoding. The `eth-rlp` library is used for their decoding. If no entry for a certain block height is found, the iteration is stopped.

Listing 4.1: Ethereum Block Body Iterator

```
 1 from __future__ import annotations
 2 from typing import Union
 3 from ethereum_freezer_tables import FreezerBodiesTable
 4 from ethereum_leveldb_tables import EthLevelDB
 5 from ethereum_rlp import Body
 6
 7 class ParseEthereumBlockBodies:
 8     def __init__(self, ancient_chaindata_path: str, chaindata_path: str)
           :
 9         self.eth_freezer_table = FreezerBodiesTable(
               ancient_chaindata_path)
10         self.eth_leveldb = EthLevelDB(chaindata_path)
11         self.value = 0
12
13     def get_body(self, number: int) -> Body:
14         try:
15             body: Union[Body, None] = self.eth_freezer_table.
                   get_body_by_height(number)
16         except:
17             body = self.eth_leveldb.get_body_by_height(number)
18         if body is None:
19             raise Exception(f"BodyNotFoundOnHeight:{number}")
20         return body
21
22     def __iter__(self) -> ParseEthereumBlockBodies:
23         return self
24
25     def __next__(self) -> Body:
26         try:
27             body = self.get_body(self.value + 1)
28         except:
29             raise StopIteration
30         self.value += 1
31         return body
```

Monero

The Monero transactions are retrieved from two LMDB database tables, `txs_pruned` and `tx_indices`, as shown in Listing 4.2 (lines 2-3). First, all content in `tx_indices` is iterated over, to retrieve transaction indices used later to retrieve the full transaction (Listing 4.2 line 6). These are filled into a cache to facilitate batched processing (Listing 4.2 line 8). Once a sufficient size, the raw `tx_indices` entries are de-serialized with the Monero Python serialization library and an in-line instantiated asynchronous runtime (Listing 4.2 line 10). The first 8 bytes of the de-serialized transaction hash are then used as a key to retrieve the transaction data from the `txs_pruned` table, and specifically whatever data is in the `tx_extra` field. The transaction data is fetched in batches using Python LMDB's `getmulti` for batched reads (Listing 4.2 line 15).

Listing 4.2: Monero Transaction Parser

```
1  env = lmdb.open(self.blockchain_path, subdir=True, lock=False, readonly=True, max_dbs=10)
2  index_db = env.open_db(b"tx_indices", integerkey=True, dupsort=True, dupfixed=True)
3  tx_db = env.open_db(b"txs_pruned", integerkey=True)
4  tx_indices_cache = []
5  with env.begin(write=False) as txn:
6      for _, tx_index in txn.cursor(db=index_db):
7          tx_indices_cache.append(tx_index)
8          if len(tx_indices_cache) == 10000:
9              # Get the TxIndex struct from the database tx_index value
10             monero_tx_indices: List[xmr.TxIndex] = asyncio.get_event_loop().run_until_complete(deserialize_tx_indices(tx_indices_cache))
11             # translate the tx index back to bytes for retrieval of the full transaction
12             db_tx_hashes: List[bytes] = [monero_tx_index.data.tx_id.to_bytes(8, "little") for monero_tx_index in monero_tx_indices]
13             # Get the full transaction from the database with the transaction id bytes
14             cursor = txn.cursor(db=tx_db)
15             monero_txs_raw: List[bytes] = cursor.getmulti(db_tx_hashes)
16             cursor.close()
17             # Use monero_txs_raw and monero_tx_indices for further processing
18             ...
19             tx_indices_cache = []
```

Bitcoin

The Python Bitcoin Blockchain Parser's `get_ordered_blocks` method first reads Bitcoin's LevelDB block height index and then uses the index to traverse the block files similarly as described before for Ethereum. UTXOs are parsed with a separate iterator developed from a customized version (typed and updated to Python3) of the STATUS library. In Bitcoin's LevelDB database UTXOs containing standard P2SH, P2PK, and P2PKH outputs are stored in a compressed state by exchanging their `OP_CODE`s for a single identification byte. Additionally, all UTXOs are stored obfuscated to avoid anti-virus software false positives [63]. Listing 4.3 shows the retrieval of the obfuscation key (line 6). Finally, the iterator wraps a LevelDB iterator and on each step de-obfuscates and de-serializes a single UTXO (Listing 4.3 line 10-15).

Listing 4.3: Bitcoin UTXO parser

```
1      # Open the LevelDB
2      db = plyvel.DB(
3          str(path.expanduser()) + "/" + fin_name, compression=None)
4      ...
5      # Load obfuscation key (if it exists)
6      self._o_key = db.get((unhexlify("0e00") + b"obfuscate_key"))
7      ...
8      self._iterator = db.iterator(prefix=prefix)
9  def __next__(self) -> Dict[str, Any]:
10     key, o_value = self._iterator.__next__()
11     key = hexlify(key)
12     if self._o_key is not None:
13         value = deobfuscate_value(self._o_key, hexlify(o_value))
14     else:
15         value = hexlify(o_value)
16     return decode_utxo(value, key)
```

4.2.2 Data Extractor Implementation

An instantiated data extractor component connects the transaction parser iterator to the database while implementing a unique set of pre-processing criteria for each blockchain. The data extractors have a common interface as defined by the abstract base class in Listing 4.4.

Each extractor is initialized with a path to its respective blockchain data files and an enumerator for the blockchain it represents. The `parse_and_extract_blockchain` abstract method takes the database adaptor as an argument. Its implementation and instantiation in sub-classes run the actual parsing, pre-processing, and writing. Unique extractors are implemented for each Bitcoin, Monero and Ethereum deriving from the `DataExtractor` abstract base class. Using a common interface across the different data extractors allows easier switching between blockchains at the call site of the extractor.

Listing 4.4: Data Extractor Base Class

```
from abc import ABC, abstractmethod
from pathlib import Path
from database import BLOCKCHAIN, Database

class DataExtractor(ABC):
    @abstractmethod
    def __init__(self, blockchain_path: Path, coin: BLOCKCHAIN) -> None:
```

```python
        pass
    @abstractmethod
    def parse_and_extract_blockchain(self, database: Database) -> None:
        pass
```

It should be noted that the runtime speed of the data extractor is mostly dominated by interactions with the filesystem. Reading blockchain data and writing to the blockchain-parser's database are both necessarily synchronous, non-concurrent operations, constrained by both locks on the various databases and the need to retain block height order. However, since the blockchain files and databases, and the blockchain-parser SQL database are distinct from each other, the data extracting and writing can be split into two concurrent tasks. Thus, to speed up parsing, filtering, and writing the results to disk, writing to the SQL database is split into a separate worker thread, while the main thread parses, extracts, and filters the blockchain data. Database writes are additionally batched.

The worker thread communicates with the main thread with the help of ZeroMQ message passing. Listing 4.5 shows an example ZeroMQ setup for the communication between the main and worker thread implemented for Ethereum, using the Pair pattern and in-process communication as the transport layer. The ZeroMQ `recv_pyobj` method blocks until a message is received.

Listing 4.5: ZeroMQ setup

```python
import zmq
database_event_sender = context.socket(zmq.PAIR)
database_event_receiver = context.socket(zmq.PAIR)
database_event_sender.bind("inproc://ethereum_dbbridge")
database_event_receiver.connect("inproc://ethereum_dbbridge")
... # a later point when iterating through the raw ethereum data
database_event_sender.send_pyobj(EthereumDataMessage(tx.data, tx.hash(),
    DATATYPE.TX_DATA, height))
... # in the worker thread:
while True:
    message: EthereumDataMessage = self._receiver.recv_pyobj()
    ... # process and write to the database
```

For each scanned historical Bitcoin block, the therein contained transactions are deserialized with `python-bitcoinlib` and each of the transaction inputs and outputs are selected. All transactions containing non-P2SH standard transaction inputs are ignored. This check includes all P2PK, P2PKH, P2SH(P2MS), P2SH(P2WPKH), and P2WPKH inputs. Additionally, all standard transaction outputs except for those containing `OP_RETURN` are ignored, including P2PKH, P2PK, P2SH, P2MS, P2WPKH, and P2WSH. A detector function for a P2SH output is included in Listing 4.6. This method has the downside of also excluding all pay-to-fake-key style data embedding methods. Since these fake outputs are unspendable though, they can be collected from the UTXO set. To this end, all UTXOs as parsed from the UTXO set are saved to the database, without additional pre-filtering applied.

Listing 4.6: P2SH output detector

```python
def is_p2sh_output(script: bitcoin.core.script.CScript) -> bool:
    """Checks if the output script is of the form:
            OP_HASH160 <hash> OP_EQUAL
    :param script: Script to be analyzed.
    :type script: bitcoin.core.script.CScript
    :return: True if the passed in bitcoin CScript is a p2sh output
        script.
    :rtype: bool """

    if len(script) != 23:
        return False
    return (
        script[0] == bitcoin.core.script.OP_HASH160
        and script[-1] == bitcoin.core.script.OP_EQUAL
    )
```

For Ethereum, transactions with empty data fields are ignored. Ethereum Request for Comments 20 (ERC-20) [65] contract calls are unlikely to hold generic data while making up a significant portion of Ethereum transactions. They are identified by their *methodId*, length, and amount of leading zero bytes in the address encoding. These include the ERC-20 `transfer`, `approve` and `transfer from` *methodIds*. Listing 4.7 shows the `transfer from` method detector, first checking the correct method id and then the presence of the addresses used as the arguments of the `transfer from` invocation.

Listing 4.7: ERC-20 `transfer from` detector

```python
ERC20_TRANSFER_FROM_METHOD_ID = bytes.fromhex("23b872dd")
ETH_LEADING_12_ZERO_BYTES = bytes.fromhex("0"*24)

def detect_erc20_transfer_from(tx_data: bytes) -> bool:
    """"Checks if the provide tx_data contains the ERC20 transfer from
        function invocation
    :param tx_data: The tx data to be analyzed
    :type tx_data: bytes"""

    # transferFrom(address _from, address _to, uint256 _value)
    if tx_data[0:4] == ERC20_TRANSFER_FROM_METHOD_ID:
        # the length of this contract call is exactly 100 bytes
        if len(tx_data) != 100:
            return False
        # check that the addresses are present, by checking the number
            of zeroes
        if not (tx_data[4:16] == ETH_LEADING_12_ZERO_BYTES and tx_data
            [36:48] == ETH_LEADING_12_ZERO_BYTES):
            return False
        return True
    return False
```

In the Monero Data Extractor, all transactions are ignored with `tx_extra` fields containing a single encrypted payment ID with a single transaction public key. Listing 4.8 shows how first the correct length of such a `tx_extra` entry is asserted followed by checks of the correct flag bytes. The Monero transaction parser has an additional runtime performance bottleneck from the required asynchronous asyncio runtime. To minimize its effect on the data processing and analysis performance, an additional worker thread is created for the sole purpose of deserializing the transactions. It accepts ZeroMQ messages from the main thread and passes its results directly by another ZeroMQ message to the database writer thread.

Listing 4.8: Monero default Transaction Extra detector

```python
def is_default_extra(extra: bytes) -> bool:
    """Checks if the tx_extra follows the standard format of:
            0x01 <pubkey> 0x02 0x09 0x01 <encrypted_payment_id>
    :param extra: Potential default extra bytes.
    :type extra: bytes
    :return: True if the passed in bytes are in the default tx_extra
       format
    :rtype: bool
    """

    if len(extra) != 1 + 32 + 1 + 1 + 1 + 8:
        return False
    if (
        extra[0] == 0x01
        and extra[33] == 0x02
        and extra[34] == 0x09
        and extra[35] == 0x01
    ):
        return True
    return False
```

4.2.3 Database Adaptor Implementation

The database adaptor provides functions for creating tables, inserting data, and querying records. Writes to the SQLite3 database are additionally batched where possible with the help of Python SQLite3's `executemany`, which combines multiple rows into a single SQL `INSERT` statement. For the analysis component, a function is provided that takes a media type detector function as an argument to analyze the data. The interface of this function is defined in Listing 4.9. Since reading and writing to the same SQLite3 database cannot be executed in parallel due to the database's read and write locks, a separation into worker threads or even separating the iterating read database cursor from the write database cursor proved architecturally challenging and was not implemented.

Listing 4.9: Media Detection Function Interface

```python
class DetectorPayload(NamedTuple):
    txid: str
    data_type: str
    extra_index: int
    data: bytes

DetectorFunc = Callable[[DetectorPayload], Optional[NamedTuple]]
DatabaseWriteFunc = Callable[[[Iterable[Any], SQLite3.Connection]]]

```

```python
def run_detection(self, detector: DetectorFunc, database_write_func:
    DatabaseWriteFunc, blockchain: Optional[BLOCKCHAIN]) -> None:
```

4.2.4 Analysis Engine Implementation

When running the blockchain-parser in analyze mode, the user is given the option to select from a host of different "detectors". A detector is a function with an interface as defined in Listing 4.9. Each detector only returns its first positive finding, or `None`. Further data in each entry once a positive finding has been made is not analyzed. The implementation of ASCII string and file type detectors is described in the following paragraphs.

Two string detectors are implemented to check if any strings are embedded inside the extracted data. The `native_strings` detector iterates through bytes in the data and checks if each of them are printable characters, or rather part of Python's `string.printable` as shown in Listing 4.10 (line 18). The length of the first detected string greater than the minimum length is returned in the `detected_data_length` field of the returned tuple.

Listing 4.10: Native ASCII detector

```python
1  class DetectedAsciiPayload(NamedTuple):
2      txid: str
3      data_type: str
4      extra_index: int
5      detected_data_length: int
6
7  def native_strings(detector_payload: DetectorPayload, min: int = 10) ->
       Optional[DetectedAsciiPayload]:
8      """Find a string with the specified minimum size
9      :param detector_payload: Contains data to be examined.
10     :type detector_payload: DetectorPayload
11     :param min: Minimum length of the to be detected string.
12     :type min: int
13     :return: DetectedAsciiPayload if detected, None if not.
14     :rtype: Optional[DetectedAsciiPayload]
15     """
16     result = ""
17     for c in detector_payload.data:
18         if chr(c) in string.printable:
19             result += chr(c)
20             continue
21         if len(result) >= min:
22             return DetectedAsciiPayload(detector_payload.txid,
                   detector_payload.data_type, detector_payload.extra_index,
                    len(result))
23         result = ""
24     if len(result) < min:  # catch result at EOI
25         return None
26     return DetectedAsciiPayload(detector_payload.txid, detector_payload.
           data_type, detector_payload.extra_index, len(result))
```

The `gnu_strings` detector uses the GNU `strings` [51] command-line utility to detect printable characters within the data. Its use is inspired by [59], who mention that using it directly on the blockchain databases, for example, Bitcoin's `.blk` files, will recover the ASCII strings. Calling this external utility with Python's subprocess library for every data entry is computationally expensive: For every data entry a new subprocess and file handles for handling the input and output have to be created. Listing 4.11 shows how the subprocess is opened, data is written to its standard input, results read from its standard output and the length of the detected string returned in the resulting tuple.

Listing 4.11: GNU `strings` ASCII detector

```python
def gnu_strings(payload: DetectorPayload, min: int = 10) -> Optional[DetectedAsciiPayload]:
    """Find and return a string with the specified minimum size using gnu strings
    :param bytestring: Bytes to be examined.
    :type bytestring: bytes
    :param min: Minimum length of the to be detected string.
    :type min: int
    :return: A string of minimum length min as detected in the bytestring.
    :rtype: str
    """
    cmd = "strings␣-n␣{}".format(min)
    process = subprocess.Popen( cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, stdin=subprocess.PIPE)
    process.stdin.write(payload.data)
    output = process.communicate()[0]
    output_str = output.decode("ascii").strip()
    length = len(output_str)
    if length < min:
        return None
    return DetectedAsciiPayload(payload.txid, payload.data_type, payload.extra_index, length)
```

Next to these two ASCII string detectors, two file type detectors are implemented. One uses the Python `imghdr` module to detect magic bytes of various image-related file formats. Magic bytes are typically a few bytes prepended to data to identify their format. The other uses a Python wrapper (`python-libmagic` [64]) for the `libmagic` [54] library, which is one of the main components of the better known GNU `file` [39] utility, to detect potential files from a large variety of different files. Checking magic bytes will on occasion lead to false positives. The file detectors do not check every possible sub-slice of the data for files. Instead, the data is chunked by removing all blockchain-specific serialization and script operator bytes. Each chunk is then analyzed in its entirety by the file detectors.

The `imghdr` module and `python-libmagic` library provide very similar interfaces, leading to similar implementations between them. For illustration purposes the implementation of the `imghdr` detector is shown in Listing 4.12. Some Bitcoin media embedding tools, for example the "publish-text" tool in the `python-bitcoinlib` library [62], add a padding byte before the data. If no file is found an additional attempt is made with the potential padding byte removed (Listing 4.12 line 12).

Listing 4.12: `imghdr` detector

```python
def find_file_with_imghdr(data: bytes) -> Optional[str]:
    """Find images with the help of imghdr magic numbers
    :param bytestring: Bytes to be examined.
    :type bytestring: bytes
    :return: A string with the file type
    :rtype: str
    """
    res = imghdr.what("", data)
    if res:
        return res
    # try again with a potential padding byte removed
    res = imghdr.what("", data[1:])
    return res
```

Bitcoin data is chunked with the help of `python-bitcoinlib`'s script parser. Its `CScript` object provides an iterator over each chunk of data contained in a Bitcoin script. In Listing 4.13 every non-`OP_CODE` part of the script is passed to the file detector function. Beforehand the entire data is checked for a potential file (Listing 4.13 line 5).

Listing 4.13: Bitcoin File Data Chunker

```python
1  from bitcoin.core import CScript, script
2  def bitcoin_find_file_within_script(detector_payload: DetectorPayload,
       file_detector_func: Callable[[bytes], Optional[str]]) -> Optional[
       DetectedFilePayload]:
3      cscript = CScript(detector_payload.data)
4      # try finding a file in the full script
5      res = file_detector_func(cscript)
6      if res is not None:
7          return DetectedFilePayload(detector_payload.txid,
               detector_payload.data_type, detector_payload.extra_index, res
               )
8      for op in cscript:
9          # ignore single op codes
10         if type(op) is int:
11             continue
12         if type(op) is script.CScriptOp:
13             continue
14         # try finding a file in one of the script pushdata
15         res = file_detector_func(op)
16         if res is not None:
17             return DetectedFilePayload(detector_payload.txid,
                   detector_payload.data_type, detector_payload.extra_index,
                    res)
18     return None
```

The monero-python library's `ExtraParser` [55] separates public keys and nonces from the complete data contained in the Monero transaction `tx_extra` field. Listing 4.14 is an excerpt of the function invoking the file detector on chunks of the `tx_extra` data. Each public key and nonce value is checked for potential file data. The `ExtraParser`'s parse method throws an exception containing offset location of the first offending byte if any non-standard bytes are found. In Listing 4.14 lines 15-25 the exception is caught and the data is sliced at the location of the offending byte passed to the file detector function.

Listing 4.14: Monero File Data Chunker

```python
from monero.transaction import ExtraParser
def monero_find_file_within_extra(detector_payload: DetectorPayload, file_detector_func: Callable[[bytes], Optional[str]]) -> Optional[DetectedFilePayload]:
    extra_data = ExtraParser(detector_payload.data)
    probable_data_index = 0
    ...
    # check every first and second byte in the nonces
    try:
        parsed_extra = extra_data.parse()
        if "nonces" in parsed_extra.keys():
            for nonce in parsed_extra["nonces"]:
                res = file_detector_func(nonce)
                if res is not None:
                    return DetectedFilePayload(detector_payload.txid, detector_payload.data_type, detector_payload.extra_index, res)
    ...
    except ValueError as err:
        # get the offset of the non-standard tx extra data
        match = get_monero_offset_regex().match(str(err))
        if match is None:
            pass
        else:
            if match.group(1) is not None:
                probable_data_index = int(match.group(1))
            res = file_detector_func(detector_payload.data[probable_data_index:])
            if res is not None:
                return DetectedFilePayload(detector_payload.txid, detector_payload.data_type, detector_payload.extra_index, res)
    ...
```

No additional data chunking is implemented for Ethereum. Nonetheless, a separate file detector implementation for Ethereum shown in Listing 4.15 lays out how the functions compose with each other. Similar function compositions with file detectors and data chunking for Bitcoin and Monero are also implemented. These composed functions are passed to the `run_detection` function of the database component as defined in Listing 4.9.

Listing 4.15: Ethereum File Detectors

```python
def ethereum_find_file_within_data(detector_payload: DetectorPayload, file_detector_func: Callable[[bytes], Optional[str]]) -> Optional[DetectedFilePayload]:
    res = file_detector_func(detector_payload.data)
    if res is not None:
        return DetectedFilePayload(detector_payload.txid, detector_payload.data_type, detector_payload.extra_index, res)
    return None

```

```
def ethereum_find_file_with_magic(detector_payload: DetectorPayload) -> Optional[DetectedFilePayload]:
    return ethereum_find_file_within_data(detector_payload, find_file_with_magic)

def ethereum_find_file_with_imghdr(detector_payload: DetectorPayload) -> Optional[DetectedFilePayload]:
    return ethereum_find_file_within_data(detector_payload, find_file_with_imghdr)
```

4.2.5 View Implementation

The results presented in the evaluation, are compiled by the view mode of the blockchain parser. It uses SQL queries to directly query histograms from the database. The query used for the ASCII text data is shown in Listing 4.16. The view prints the raw data of the histogram to `stdout` on the command line (lines 2-7), saves and shows plots of the histograms to the user, and writes the histograms to `CSV` files.

Listing 4.16: Sample SQL histogram query

```
1 SELECT STRING_LENGTH, COUNT(STRING_LENGTH) FROM asciiData GROUP BY STRING_LENGTH ORDER BY STRING_LENGTH;
2 > ppm|5
3 pbm|11
4 pgm|13
5 bmp|303
6 tiff|616
7 rgb|18002
```

4.2.6 Usage

The blockchain-parser tool provides a help text when run with the `-help` flag. Example usage within the three implemented modes, parse, analyze and view, is nevertheless provided in the following.

The blockchain-parser command in Listing 4.17 illustrates its usage in parse mode. `-parse` takes as an argument the location of the respective blockchain's data directory. `-blockchain` specifies which blockchain is supposed to be targeted, `-database` the name of the database file the parsed data is written to. Valid blockchain arguments can be read from the help text.

Listing 4.17: Blockchain-Parser Parse Mode

```
python main.py --parse /home/drgrid/.bitcoin/testnet3 --blockchain bitcoin_testnet3 --database btc_test.db
```

Listing 4.18 is an example for running the tool in analyze mode. Analyze takes as an argument an identifier for the detector function to be used. The `-help` text provides all valid analyzer arguments. While not strictly required, the blockchain argument is relevant where data chunking as illustrated in Listing 4.13 and Listing 4.14 is possible.

Listing 4.18: Blockchain-Parser Analyze Mode

```
python main.py --analyze magic_files --blockchain monero_mainnet --
    database xmr_result.db
```

Lastly, the view mode usage is shown in Listing 4.19. The view argument determines for which detected data type a histogram should be compiled. Again, the help text provides all valid data type arguments for the view mode. The data can additionally be restricted, labeled, and color-coded to specific blockchains with the `-blockchain` argument. When passed the `record_stats` flag, the view compiles and prints statistics of the data tables as shown in Listing 4.20 including the total number of rows in each table and the maximum block height found in the `blockchainData` table.

Listing 4.19: Blockchain-Parser ASCII-Histogram View Mode

```
python main.py --view ascii_histogram --blockchain monero_mainnet --
    database xmr_result.db
```

Listing 4.20: Blockchain-Parser Record-Stats View Mode

```
python main.py --view record_stats --blockchain monero_mainnet --
    database xmr_result.db
> Maximum data record size: 165035
RecordStatistics(distinct_data_rows=5475178, max_block_height=2558959,
    ascii_data_count=178366, magic_file_data_count=82108,
    imghdr_file_data_count=18950)
```

Chapter 5

Evaluation and Discussion

Each of the blockchains considered in this thesis are parsed and analyzed with the blockchain-parser tool. The blockchain-parser's view mode is then used to generate the presented histograms. Table 5.1 shows at which block height the data was recorded for each blockchain as well as the total number of rows in the raw `blockchainData` table and the tables populated by the various analysis engine detectors. Additionally, the maximum size of a single data record in bytes for each blockchain is given. The last row contains the full raw size of the respective blockchain.

Table 5.1: Data Collection Statistics

	Bitcoin	**Ethereum**	**Monero**
Block Height	725'463	10'230'173	2'558'959
Last Block Date	March 1st 2022	June 9th 2020	February 13th 2022
Raw Data Rows	234'257'580	155'271'423	5'475'178
String Rows	5'985'826	9'216'636	178'366
Magic File Rows	2'220'047	3'292'963	82'108
Imghdr File Rows	27'843	851	18'950
Max. Data Record Size (Bytes)	9319	716'805	165'035
Database Size (MBytes)	61'459	67'664	1'510
Blockchain Size (GBytes)	456	494	134

Due to time constraints in collecting data for this thesis, the block syncing from the three blockchains was stopped at different points in time. The block syncing of Ethereum was stopped prematurely after a week of real-time syncing to test and develop the blockchain-parser. Though not benchmarked accurately, estimated runtime durations are given in the following sentences. The Monero parsing and analysis steps did not exceed a few hours. Bitcoin and Ethereum's block parsing each took more than a full day. The native string and `imghdr` detector-based analyses each took about 8 hours. The file type analysis with the `python-libmagic` library in each case took about 24 hours. The Monero blockchain has recorded orders of magnitude fewer transactions than both Bitcoin and Ethereum, explaining the discrepancy in both the amount of data parsed, recorded, and analyzed, and thus the differences in analysis runtime. Once the analysis was completed, producing the results with the blockchain-parser's view was performed in seconds.

To decrease the initial blockchain download, data parsing, and analysis time, each blockchain was processed with its dedicated operating system and hardware. Parallel processing would have been possible as long as separate databases are used for each blockchain to avoid deadlock. Table 5.2 shows the system configuration used for each blockchain, including the Operating System (OS) on which the analysis was conducted on. Besides the disk type and its effect on file I/O, none of the system resources were exhausted or seemed to have negatively affected processing speed. The slower Ethereum initial blockchain sync speed can be explained by the slower read and write performance of a Hard Disk Drive (HDD) compared to a Solid-State Drive (SSD).

Table 5.2: System Information

	Bitcoin	**Ethereum**	**Monero**
CPU	Ryzen 7 3700X @ 3.6GHz	Xeon E312xx @ 2.4 GHz	i7-5820K @ 3.3 GHz
RAM	64 GB	16 GB	24 GB
Disk Size	921 GB	2047 GB	464 GB
OS	Ubuntu 18.04	Ubuntu 20.04	Ubuntu 18.04
Disk Type	SSD	HDD	SSD

5.1 Histograms

The ASCII string length histograms depict the string length distribution of detected strings with a minimum length of 10. For each blockchain, a histogram showing the distribution of string length counts and the total strings above a certain string length is generated. The x-axis in each plot is logarithmically scaled.

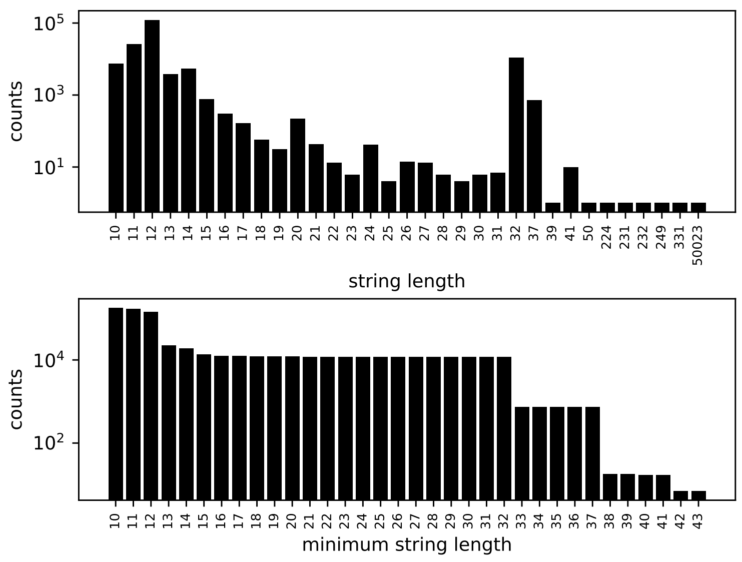


Figure 5.1: Monero ASCII String Logarithmic Histogram

The entire distribution of Monero string lengths is plotted in Figure 5.1. The single outlier in the 50'023 string length bin was inspected closer and contains the phrase "Improve uniformity, remove tx_extra." repeated over and over again in the `tx_extra` field of the transaction with TXID 108E...791E. The limitations of the ASCII string detection method used by the blockchain-parser are illustrated by the content in the 249 string length bin. On inspection, it contains the full script of DreamWorks Animation's "Bee Movie", a string with length 115'345 contained in the `tx_extra` field of the transaction with TXID 50DE...1478. The difference between the actual and detected string length is explained by the inclusion of special non-ASCII characters in the data. The string detector of the blockchain-parser will only detect the first ASCII substring longer than 10 characters before the special non-ASCII characters are found. Indeed the first special character appears 249 characters after the beginning of the first string. Prepended and appended to the movie script is a message again calling for the immediate removal of the `tx_extra`. Further inspection of the strings reproduces the existing findings in [38], including E-Mail addresses, greetings, and short texts.

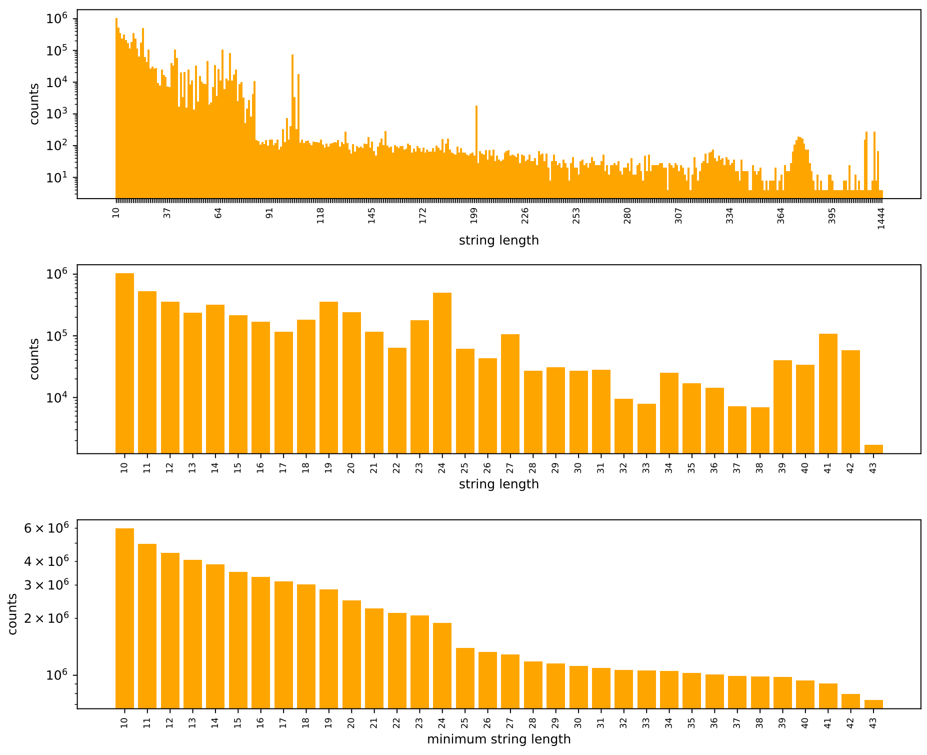


Figure 5.2: Bitcoin ASCII String Logarithmic Histogram

The top histogram of the Bitcoin string length Figure 5.2 shows the distribution of string lengths in its entirety. The labels added to its bins offer coarse orientation to their representing string length, however, empty bins are skipped to visualize their entire distribution. The last bin with string length 1'444 is a string from a non-standard transaction output in the transaction with TXID 7782...AD69 containing an E-Mail supposedly written by

Satoshi Nakamoto warning a Bitcoin implementation of a security bug. Several outliers can be distinguished in the data, with a cluster of strings appearing around the 64-byte string length. For visualization and comparison purposes only the first 33 bins are plotted in the second plot of Figure 5.2. Even so, outliers can still be identified for example in bin 24. During the development of the blockchain-parser, some of the detected Bitcoin strings were viewed closer, confirming prior research on their content containing, among others, short poems, greetings, and E-Mail addresses.

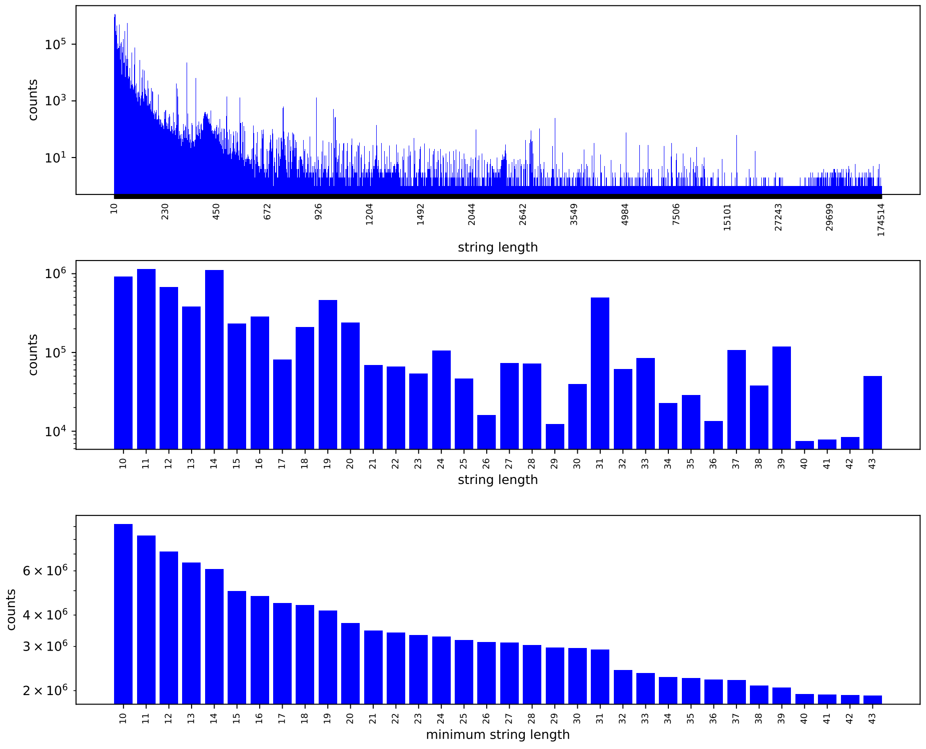
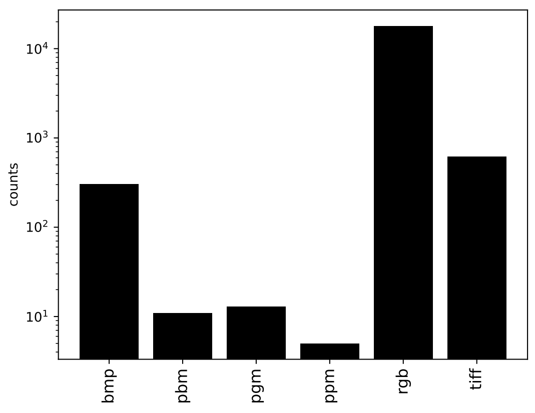


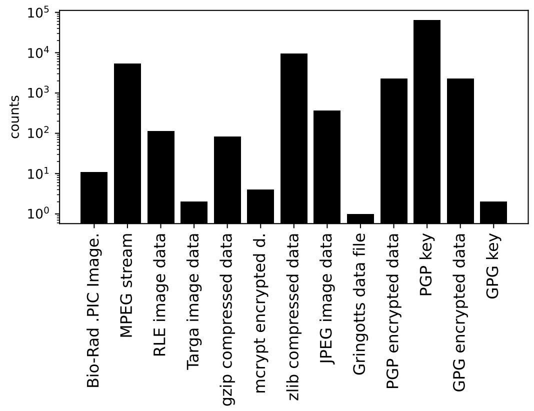
Figure 5.3: Ethereum ASCII String Logarithmic Histogram

Similar to Bitcoin's, the top histogram in the Ethereum string length Figure 5.3 omits empty bins. The final bin with string length 174'514 corresponds to the data field content of the Ethereum transaction with TXID 2F05...D9AE containing a full copy of William Shakespeare's Romeo and Juliet as published by Project Gutenberg [58]. A cluster around the 450 string length bin can be visually identified. The second strings histogram in Figure 5.3 is also truncated to the first 34 bins. Its data seems more evenly distributed, with just a single, clear visual outlier in bin 31. From the few other strings beheld during development, their content seemed similar to that of Bitcoin and Monero.

The `python-libmagic`- and `imghdr`-based file type detector results are histogrammed within the same figure for each blockchain. Comparing the histograms visually, a few prevalent file types seem common across all blockchains. Again, the counts in each histogram are logarithmically scaled. When comparing the distribution of `python-libmagic`-detected file types between the blockchains, a prevalence for `UTF-8`, `JSON` and `CSV` data can be identified in Ethereum. The distributions of Monero's detected file type histograms presented in Figure 5.4a and 5.4b seem like a subset of the detected file types in the histograms of Bitcoin and Ethereum with no apparent file types sticking out, besides a high count of `rgb` type images.



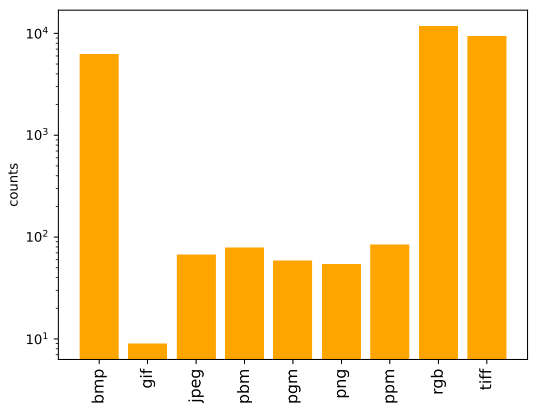
(a) Monero `imghdr`-detected file types



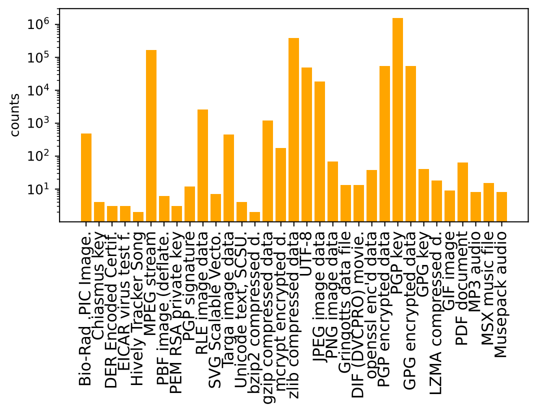
(b) Monero `python-libmagic`-detected file types

Figure 5.4: Monero detected file types logarithmic histogram

Some of the `imghdr`-detected `jpeg` file types found in Bitcoin and presented in Figure 5.5a were manually inspected. Their transaction IDs matched the ones presented in the prior work of [59]. However, no complete reconstruction of an image was attempted, since further tooling would have been required to retrieve other parts of the image from other transaction outputs or even other transactions. Due to time limitations, this was not developed.



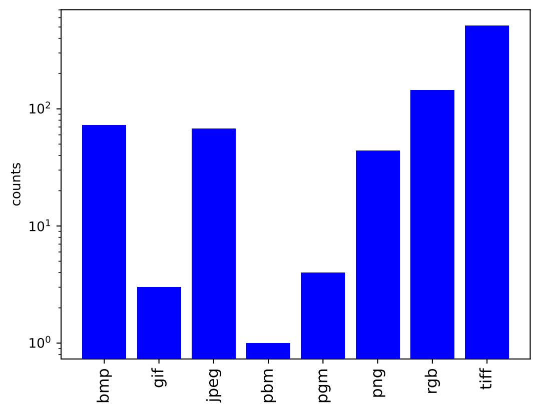
(a) Bitcoin `imghdr`-detected file types



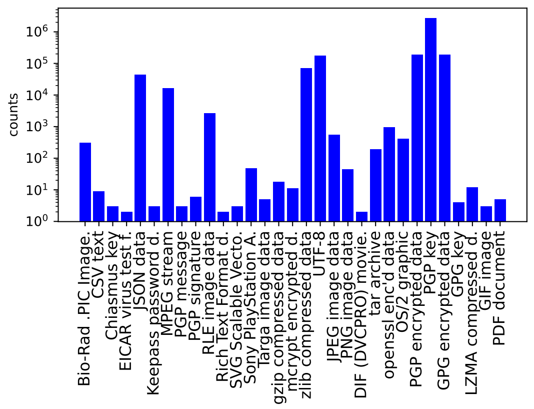
(b) Bitcoin `python-libmagic`-detected file types

Figure 5.5: Bitcoin detected file types logarithmic histogram

The three potential `gif` type files detected from Ethereum and represented in the data of Figure 5.6a were also manually inspected. Their entire raw data was retrieved from



(a) Ethereum `imghdr`-detected file types



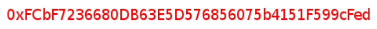
(b) Ethereum `python-libmagic`-detected file types

Figure 5.6: Ethereum detected file types logarithmic histogram

the database and written to a file. Indeed each of the data contained a complete, single-framed `gif` file, Figure 5.7a showing an extracted QR code, Figure 5.7b a "troll face" and Figure 5.7c what appears to be the first 20 bytes of an Ethereum private key. Due to time constraints, no other potential files were inspected.

(a) TXID 7DE..3A2

(b) TXID CC44..430E


0xFCbF7236680DB63E5D576856075b4151F599cFed

(c) TXID C9F6..BE23

Figure 5.7: `GIF` files found in Ethereum

5.2 False Positive Detections

Much of the data presented in the histograms are made up of false positives. The likelihood of finding an ASCII string of length n within a series of uniformly distributed bytes *decreases* exponentially with the length of n. In the ASCII string histograms, the number of detected strings is thus skewed towards the left-hand side. When visualized on a logarithmic axis the string length appears to decrease linearly within the first few bins. Outliers of this pattern are where in all likelihood actual strings were detected. A concrete calculation of the false positive rate for detected ASCII strings is complex. Though the probability of detecting a single character in a byte of data is $p = 100/256$ for the 100 printable characters contained in Python's `string.printable`, calculating the probability of detecting a string with a minimum length n is dependent on the string's length, which varies for each record. Thus, such a calculation is omitted here.

The likelihood of a false positive file type identification is based on the length and restrictions of the file type's magic bytes. As an example, the magic bytes of one of the most often `imghdr`-detected file type, `rgb`, are `\x01\xda`. The probability of drawing these two bytes at the first and second position of the data respectively is $p = 256^{-2}$ and the expected number of false positives for n trials, assuming the bytes in the data are randomly distributed, $n \cdot p$. Calculating the expected false-positive rate on the data here is however more involved since the data is indeed not randomly distributed. However, what should be noted is that file types identified by only 2 or 3 magic bytes are in all likelihood false positives.

Table 5.3: Expected `imghdr` File Detection Events

File Type	l	n	p	**EE Monero**	**AE Monero**	**EE Bitcoin**	**AE Bitcoin**	**EE Ethereum**	**AE Ethereum**
`tiff`	2	2	$2 \cdot 256^{-2}$	668	616	28595	9434	9477	514
`rgb`	2	1	256^{-2}	334	18'002	14279	11796	4738	144
`bmp`	2	1	256^{-2}	334	303	14279	6261	4738	73
`pbm`	3	6	$6 \cdot 256^{-3}$	11	11	335	79	111	1
`pgm`	3	6	$6 \cdot 256^{-3}$	11	13	335	59	111	4
`ppm`	3	6	$6 \cdot 256^{-3}$	11	5	335	84	111	0
`jpeg`	4	2	$2 \cdot 256^{-4}$	0	0	0	67	0	68
`gif`	6	2	$2 \cdot 256^{-6}$	0	0	0	9	0	3
`png`	8	1	256^{-8}	0	0	0	54	0	44

Table 5.3 shows the expected numbers of detected data assuming a uniform distribution of its bytes from a few selected file types. First, the probability p for a single false positive is calculated from the magic bytes. This probability is multiplied by a value estimating the number of k data chunks for each blockchain. For Monero this is estimated at 6, including a transaction public key, a payment ID, and the entire data with each padding byte removed and left whole. Bitcoin transaction input or outputs usually contain at least one data element, so its k is estimated to 4, while Ethereum data is not chunked any further beyond the padding byte, making its $k = 2$. A certain file type is marked by n different magic bytes of length l. Finally the total number of records r is retrieved from Table 5.1. The expected number of detected file types, or Expected Events (EE), is then $n \cdot p \cdot k \cdot r$. These are analog to the expected number of false positives. Additionally the Table 5.3 gives the number of Actual Events (AE) for each detected file type.

From comparing the expected to the actual number of detected file types most if not all of the detected `tiff`, `rgb`, or bitmap files are false positives. However, any detected `jpeg`, `gif`, and `png` file types may represent real files with a high likelihood. Considering that these are three widely used image file formats adds evidence to this hypothesis. Based on the difference between the estimated and actual data in Bitcoin and Ethereum, their k value seems to have been chosen too high.

Similar to the `imghdr`-detected file types, the `python-libmagic`-detected file type results also contain many false positives. Comparing the results among each other, all blockchains contain similarly large amounts of PGP data, an indication that these are further false positives, especially when comparing their counts to the counts of detected `png`. The `gif` and `png` image file types are also detected by both `imghdr` and `python-libmagic`, contrary to `jpeg` identification, where the `python-libmagic` detector seems to produce many false positives. The `python-libmagic` file detector seems especially well suited for identifying structured data, *e.g.,* `CSV`, `UTF-8` and `JSON` data. For these file types, the distribution between the blockchains varies significantly. Some of their identifications in Ethereum were checked manually and indeed all contained the structured data that `python-libmagic` claimed they did.

5.3 Discussion and Challenges

This section recapitulates the results of the evaluation, discusses them, and briefly contextualizes them with those of some related works. Finally, it presents challenges faced during implementation and data gathering.

5.3.1 Discussion

In the Monero ASCII string length histogram of Figure 5.1 a clear outlier is in the 32-byte bin. These are likely small messages embedded in the 32-byte unencrypted payment IDs [38]. No explanation was found for the apparent clusters in the Bitcoin and Ethereum string data. For each blockchain, the aggregate minimum string length bins are plotted to discover bins that significantly deviate from the baseline. Such can be found for string length 37 in Monero, 24 in Bitcoin, and 31 in Ethereum. Ascertaining if these outliers contain actual strings and not false positives just by their amount compared to other bins is however problematic. They may be caused by `OP_CODE`s in the script contracts whose byte representations lie within the printable ASCII character range. Analyzing the strings with a language engine could bring more certainty.

An example indicating that the bytes of the data analyzed here are not composed of uniformly distributed bytes can be seen in the Monero `imghdr`-detected file type histogram Figure 5.4a. If the bytes in the data were indeed uniformly distributed, the expected likelihood of a `tiff` false positive should be twice as high as a `rgb` false positive file type identification as shown in Table 5.3. The discrepancy in the data can be explained by noting that \`x01`, the first magic byte of the `rgb` file type, is used at the beginning of most Monero `tx_extra` data as a serialization marker byte indicating the presence of a transaction public key. Analog explanations also apply to the overestimation of expected file detection events for Bitcoin and Ethereum in Table 5.3.

The capabilities of the blockchain-parser to correctly identify at least a subset of the files and texts embedded into blockchains were proven by extracting and inspecting, albeit manually, some image and text data. To further increase its utility both its accuracy and speed should be improved. The ASCII-string detector should handle at least some special characters to not terminate its counting early. File and data carving tools, for example, disk recovery utilities, could potentially be used for efficient file extraction once a file has been identified. Eventually, the blockchain-parser should be able to automatically extract media files from blockchains. Though the blockchain-parser does not implement transaction clustering that would be required to extract files spread across multiple Bitcoin transactions, this could be sidestepped by using a block explorer or Bitcoin node API to assemble the required transactions. Clustering all transactions of a blockchain would be wasteful compared to assembling the required data ad-lib for the couple of hundred identified potential files.

From Table 5.1 it appears that Ethereum is the most-used blockchain for data embedding of the three, both in absolute and relative terms. Though it has the least amount of `imghdr`-detected data, this is mostly due to the low amount of likely false positives. When comparing the probably correctly identified data types, `gif`, `jpeg` and `png`, it has the most records for each. The reason for the low rate of `imghdr` detections might be that the various *methodId*s embedded in the data field have no common bytes with the magic numbers of the tested file types. A possible explanation for why Ethereum contains the most data is that it is cheaper. However, such an assessment is difficult, since the gas and Ether prices both fluctuate. A more likely explanation is that many Ethereum wallets allow their users to manually fill the transaction data field, thus allowing them to embed strings and files.

Compared to the media statistics provided by [43] in Table 3.1 the blockchain-parser discovered many more text and media records. This can be partly explained by the inclusion of false positive entries and the later blockchain height at which the analysis here was commenced from. However, a closer comparison or contextualizing of the results is not possible, since [43] does not disclose their methods for detecting data. [59] identified 129'410 unique P2FKH outputs storing ASCII strings with a minimum length of 18. In total the blockchain-parser detected 3'024'579 strings with a minimum length of 18 (Figure 5.2). This shows that many strings are stored with data embedding methods other than P2FKH. [59] also provides TXIDs of transactions containing images that appeared in the Bitcoin detected media data generated by the blockchain-parser.

5.3.2 Implementation Challenges

The main implementation challenge was reducing the computation time required in the parse and analyze steps of the blockchain-parser. Even with all optimizations outlined in Section 4.2, some of the parsing and analysis steps took over 24 hours. This made debugging particular issues that only arose deep within the respective processes, e.g. system resources, caching, and data type exceptions, challenging and time-consuming. One such issue was a memory leak in the `libmagic` library, which was only discovered after a full day of processing. Due to time constraints on this thesis, the blockchain-parser was not further optimized. Another key optimization might be batching blockchain file reads, by reading ahead in the iterator and only processing chunks of the data in each iteration. Further, the data written to the blockchain-parser's database can be reduced by placing duplicate items, like duplicate block heights, or transaction IDs, into separate tables.

The `python-libmagic` file type detector created a lot of false positive and noisy output. The noisy output was owned to `python-libmagic` creating custom results based on potential file sizes, encryption modes, and stream modes among others. These results were labeled together by matching common substrings, for example, any `python-libmagic` result containing the string "MPEG" was re-labeled as "MPEG stream" by the analyzer in the database. Further, highly probable false positive results, *e.g.* esoteric or outdated file types, were manually filtered and ignored by matching substrings.

No data analysis was made with the GNU `strings`-based detector because its runtime was too slow for the amount of data analyzed. Two attempts were made to improve its performance. In the first attempt, a strings subprocess was launched asynchronously and awaited in batches in a dedicated thread in order not to block the database reading and writing. However, Python's asynchronous subprocess had limitations when used in a thread outside the main thread, which rendered this approach too architecturally challenging. In the second attempt, many subprocesses were launched in dedicated threads. This proved much faster until the system ran out of file handles opened by the input and output streams to capture the results, leading to a crash of the program. No workaround was found to these problems, leading to the adoption of a single-threaded implementation.

Chapter 6

Summary, Conclusions, and Future Work

The ever-increasing propensity for posting and storing digital media online has left its mark on blockchains. However, their capability of storing data is limited by size restrictions, consensus rules, and transaction fees. Even with these restrictions and against further odds like complex methodologies and discouragement by blockchain developers users have found ways to embed data in blockchains. The objective of this thesis was to analyze both the amount of data and the types of media embedded into the Bitcoin, Ethereum, and Monero blockchains. Comparing results between the blockchains can then provide pointers towards which blockchain is most suited for data storage and how future blockchain-based storage approaches can cater to the needs of their users.

To provide a theoretical background needed to develop a solution to parse and analyze blockchain data, a conceptual introduction to blockchains was given. For each of Bitcoin, Ethereum, and Monero their core features and functionality, transaction structures, blockchain database designs, and data embedding methods were illustrated. Existing methods for embedding, retrieving, and analyzing data were surveyed from related work. From the related work, a lack of a cross-blockchain solution as well as quantitative analyses of blockchain-embedded media data, was identified. Finally, their description of methods and tools employed informed the development of a solution for this thesis.

As a software solution, the blockchain-parser tool was developed for identifying embedded media in blockchains. It directly parses the blockchain database and writes to a database with a generic schema. From there a user can direct the tool to analyze the parsed data for either file types or strings using multiple tools. The user can then view the analysis results through either the blockchain-parser or a query to its SQL database. The user can dynamically select blockchains and their data directories.

With the help of the blockchain-parser, ASCII text and file type statistics were compiled. For each blockchain, the tool identified text data and for Bitcoin and Ethereum some of the embedded media files. A few of the detected strings and files were successfully extracted from the blockchain-parser database and presented in this thesis. The number of expected data type identifications was compared to those of file types actually identified. This provided strong evidence that a lot of the detected files, mostly those identified through short 2- or 3-byte magic numbers, were false positives. Due to time limitations, no automated file extraction was implemented, though some files were extracted manually as proof of the method.

In conclusion, the Ethereum blockchain contained the most embedded text and media data, though all blockchains were indeed used to embed generic media. This corroborates the claim that blockchain storage is alluring for users, despite not being strictly designed, nor particularly efficient, for this purpose. Blockchain storage seems to be especially attractive for small images and texts. A system seeking to emulate and replace blockchain data storage should probably strive to be a public, uncensorable, permanent, but size-limited global bulletin board with a single up-front cost.

Of the three investigated blockchains, Ethereum seems to be the best suited for embedding generic data due to the ease of use of manipulating its data field, the available space for consecutive data, and the least detrimental effects to the network. Ethereum compresses and archives the data contents of EOA-to-EOA transactions and unlike some of the data-embedding methods of Bitcoin and Monero does not cache, or save this data to more memory-intensive databases. Although Monero also allows for large amounts of consecutive data, it is probably less suited for data storage due to its detrimental effects on transaction uniformity and thus the privacy of its users.

The thesis has left out some aspects of embedding data across different blockchains that can be followed up in future work. No complete study was done on the user experience of embedding data in each blockchain, nor the concrete associated cost for both the user in terms of transaction fee and the network in terms of transaction size and processing cost. Besides this, more blockchains could be investigated as the developed system is blockchain-agnostic. This work compared a single example of each of the UTXO-, TXO- and Account-based transaction models. Other blockchains implementing each of these transaction models could be analyzed for further comparison. The capabilities of the blockchain-parser could be improved. Most importantly, it should eventually get the ability to extract and save files from the supported blockchains. Both analysis speed and precision are areas that can be the subject of future work investigations and development.

Though mentioned both in the background and related work chapters, no steganalysis was done as part of this thesis, in part due to the lack of automated file extraction. Once implemented, future work can implement a steganalysis approach for detecting LSB-replacement in the extracted images with the herein discussed techniques, such as monobit failure detection and Shannon entropy calculations.

Bibliography

[1] Gavin Andresen. Block v2, Height in Coinbase, June 2012. `https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki`, Last visit October 25 2021.

[2] Andreas M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain.* O'Reilly Media, 2nd edition, June 2017.

[3] Andreas M. Antonopoulos. *Mastering Ethereum: Building Smart Contracts and DApps.* O'Reilly Media, 1st edition, December 2018.

[4] Massimo Bartoletti, Bryn Bellomy, and Livio Pompianu. A journey into bitcoin metadata. *Journal of Grid Computing*, 17(1):3–22, 2019.

[5] Massimo Bartoletti and Livio Pompianu. An analysis of bitcoin op_return metadata. *Lecture Notes in Computer Science*, February 2017.

[6] BeckyMH. OpenPuff, December 2017. `https://en.bitcoinwiki.org/wiki/OpenPuff`, Last visit October 31 2021.

[7] Marianna Belotti, Nikola Božić, Guy Pujolle, and Stefano Secci. A Vademecum on Blockchain Technologies: When, Which, and How. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838, 2019.

[8] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, and Francesco Tiezzi. Analysis of ethereum smart contracts and opcodes. In *International Conference on Advanced Information Networking and Applications*, pages 546–558. Springer, 2019.

[9] Stefano Bistarelli, Ivan Mercanti, and Francesco Santini. A suite of tools for the forensic analysis of bitcoin transactions: Preliminary report. In *European Conference on Parallel Processing*, pages 329–341. Springer, 2018.

[10] Stefano Bistarelli, Ivan Mercanti, and Francesco Santini. An analysis of non-standard transactions. *Frontiers in Blockchain*, 2, August 2019.

[11] bitcoin.org. What is a full node?, October 2021. `https://bitcoin.org/en/full-node#what-is-a-full-node`, Last visit October 25 2021.

[12] Benedikt Boehm. Stegexpose - a tool for detecting lsb steganography. *arXiv preprint arXiv:1410.6656*, 2014.

[13] Christian Cachin. An information-theoretic model for steganography. In David Aucsmith, editor, *Information Hiding*, pages 306–318, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[14] Antoine Le Calvez. python-bitcoin-blockchain-parser, November 2015. `https://github.com/alecalve/python-bitcoin-blockchain-parser`, Last visit January 25 2022.

[15] Jason Carver. Ethereum recursive length prefix encoding, January 2018. `https://github.com/ethereum/eth-rlp`, Last visit March 10 2022.

[16] Rajarathnam Chandramouli, Mehdi Kharrazi, and Nasir Memon. Image steganography and steganalysis: Concepts and practice. In Ton Kalker, Ingemar Cox, and Yong Man Ro, editors, *Digital Watermarking*, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[17] Howard Chu. Lmdb, June 2011. `https://git.openldap.org/openldap/openldap/tree/mdb.master`, Last visit March 10 2022.

[18] Resul Das. An investigation on information hiding tools for steganography. *International Journal of Information Security Science*, 3(3):200–208, 2014.

[19] Sergi Delgado-Segura, Cristina Pérez-Sola, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. Analysis of the bitcoin utxo set. In *International Conference on Financial Cryptography and Data Security*, pages 78–91. Springer, 2018.

[20] Maya Dotan, Yvonne-Anne Pignolet, Stefan Schmid, Saar Tochner, and Aviv Zohar. Survey on cryptocurrency networking: Context, state-of-the-art, challenges. *arXiv preprint arXiv:2008.08412*, 2020.

[21] embiimob. apertus, December 2013. `http://apertus.io/`, Last visit February 28 2022.

[22] eternitywall. eternitywall. `https://eternitywall.it/`, Last visit February 28 2022.

[23] Open Ethereum. Proof-of-authority chains - wiki, October 2021. `https://openethereum.github.io/Proof-of-Authority-Chains`, Last visit October 30 2021.

[24] Maureen Farrell. How porn links and ben bernanke snuck into bitcoin's code, March 2013. `https://money.cnn.com/2013/05/02/technology/security/bitcoin-porn/index.html`, Last visit January 30 2022.

[25] Abba Garba, Zhi Guan, Anran Li, and Zhong Chen. Analysis of man-in-the-middle of attack on bitcoin address. In *ICETE (2)*, pages 554–561, 2018.

[26] Jeff Garzik. On bitcoin data spam, and evil data, April 2013. `http://garzikrants.blogspot.com/2013/04/on-bitcoin-data-spam-and-evil-data.html`, Last visit January 30 2022.

[27] Alexandre Augusto Giron, Jean Everson Martina, and Ricardo Custódio. Steganographic analysis of blockchains. *Sensors*, 21(12):4078, 2021.

[28] Steinar H. Gunderson. snappy, June 2015. `https://github.com/google/snappy`, Last visit February 5 2022.

[29] Dwayne Richard Hipp. Sqlite3, 2000. `https://www.sqlite.org/index.html`, Last visit March 27 2022.

[30] Teng Hu, Xiaolei Liu, Ting Chen, Xiaosong Zhang, Xiaoming Huang, Weina Niu, Jiazhong Lu, Kun Zhou, and Yuan Liu. Transaction-based classification and detection approach for ethereum smart contract. *Information Processing & Management*, 58(2):102462, 2021.

[31] Storj Labs Inc. storj. `https://www.storj.io/`, Last visit February 28 2022.

[32] Sanjay Ghemawat Jeffrey Dean. Leveldb, September 2011. `https://github.com/google/leveldb`, Last visit January 28 2022.

[33] Harry Kalodner, Malte Möser, Kevin Lee, Steven Goldfeder, Martin Plattner, Alishah Chator, and Arvind Narayanan. BlockSci: Design and applications of a blockchain analysis platform. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2721–2738. USENIX Association, August 2020.

[34] Ismail Karadogan and Resul Das. An examination on information hiding tools for steganography. *International Journal of Information Security*, 3:200–208, September 2014.

[35] Dusan Klinec. Monero serialize, February 2018. `https://github.com/ph4r05/monero-serialize`, Last visit Jaunuary 28 2022.

[36] Sarang Noether Koe, Kurt M. Alonso. Zero to Monero 2nd Edition, April 2020. `https://web.getmonero.org/library/Zero-to-Monero-2-0-0.pdf`, Last visit November 2 2021.

[37] Sebastian Kung. blockchain-parser, December 2021. `https://github.com/TheCharlatan/blockchain-parser`, Last visit January 31 2022.

[38] Noncesense Research Lab. Monero tx_extra data analysis: Ascii data, August 2020. `https://github.com/noncesense-research-lab/monero_tx_extra/blob/master/ascii_data.md`, Last visit October 25 2021.

[39] AT&T Bell Laboratories. file, 1986. `http://www.darwinsys.com/file/`, Last visit March 20 2022.

[40] Protocol Labs and Juan Benet. filecoin. `https://filecoin.io/`, Last visit February 28 2022.

[41] Annick Lesne. Shannon entropy: a rigorous notion at the crossroads between probability, information theory, dynamical systems and statistical physics. *Mathematical Structures in Computer Science*, 24(3), 2014.

[42] Pieter Hintjens Martin Sustrik. Zeromq, August 2009. `https://zeromq.org/`, Last visit March 5 2022.

[43] Roman Matzutt, Jens Hiller, Martin Henze, Jan Henrik Ziegeldorf, Dirk Müllmann, Oliver Hohlfeld, and Klaus Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 420–438, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

[44] Greg Maxwell. [bitcoin-dev] capacity increases for the bitcoin system, December 2015. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-December/011865.html`, Last visit February 7 2022.

[45] Evgeny Medvedev. Ethereum etl, April 2018. `https://github.com/blockchain-etl/ethereum-etl`, Last visit January 25 2022.

[46] 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. *Commun. ACM*, 59(4):86–93, March 2016.

[47] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. An empirical analysis of traceability in the monero blockchain. *Proceedings on Privacy Enhancing Technologies*, 2018:143–163, June 2018.

[48] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, March 2009.

[49] Shen Noether. Ring Confidential Transactions, February 2016. `https://web.getmonero.org/resources/research-lab/pubs/MRL-0005.pdf`, Last visit November 2 2021.

[50] Travis Oliphant. Numpy, 2005. `https://numpy.org/about/`, Last visit March 17 2022.

[51] now licensed by the Free Software Foundation as part of GNU binutils Originally from Bell Labs. Gnu strings, 1991. `https://sourceware.org/binutils/docs/binutils/strings.html`, Last visit March 25 2022.

[52] Juha Partala. Provably secure covert communication on blockchain. *Cryptography*, 2:18, August 2018.

[53] Ouziel Slama Robby Dermody, Adam Krellenstein. Counterparty, 2013. `https://counterparty.io/`, Last visit March 30 2022.

[54] Mans Rullgard and Christos Zoulas. libmagic. `https://man7.org/linux/man-pages/man3/libmagic.3.html#LIBRARY`, Last visit March 20 2022.

[55] Michal Salaban. monero-python, November 2017. `https://github.com/monero-ecosystem/monero-python/graphs/contributors`, Last visit March 20 2022.

[56] Andreas Schildbach. bitcoinj, May 2011. `https://github.com/bitcoinj/bitcoinj`, Last visit January 30 2022.

[57] Sergi Delgado Segura. Status, November 2017. `https://github.com/sr-gi/bitcoin_tools/tree/master/bitcoin_tools/analysis/status`, Last visit February 5 2022.

[58] William Shakespeare. *The Tragedy of Romeo and Juliet.* Project Gutenberg, 1597.

[59] Andrew Sward, Ivy Vecna, and Forrest Stonedahl. Data insertion in bitcoin's blockchain. *Ledger*, 3, April 2018.

[60] tevador. Consider removing the tx_extra field, June 2020. `https://github.com/monero-project/monero/issues/6668`, Last visit November 2 2021.

[61] Harding Theymos, Belcher. Full Node, October 2021. `https://en.bitcoin.it/wiki/Full_node`, Last visit October 25 2021.

[62] Peter Todd. python-bitcoinlib, April 2011. `https://github.com/petertodd/python-bitcoinlib`, Last visit January 31 2022.

[63] Wladimir J. van der Laan. Obfuscate database files #6613, September 2015. `https://github.com/bitcoin/bitcoin/issues/6613`, Last visit March 10 2022.

[64] Dima Veselov. python-libmagic, 2014. `https://github.com/dveselov/python-libmagic`, Last visit March 20 2022.

[65] Fabian Vogelsteller. Erc-20, November 2015. `https://eips.ethereum.org/EIPS/eip-20`, Last visit February 10 2022.

[66] Nicolas von Saberhagen. CryptoNote v2.0, October 2013. `https://bytecoin.org/old/whitepaper.pdf`, Last visit November 2 2021.

[67] Benjamin Wallace. The rise and fall of bitcoin, November 2011. `https://www.wired.com/2011/11/mf-bitcoin/`, Last visit January 30 2022.

[68] Wenbo Wang, Dinh Thai Hoang, Peizhao Hu, Zehui Xiong, Dusit Niyato, Ping Wang, Yonggang Wen, and Dong In Kim. A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks. In *IEEE Access*, volume 7, pages 22328–22370, January 2019.

[69] Nic Watson. Python lmdb, February 2013. `https://github.com/jnwatson/py-lmdb`, Last visit January 28 2022.

[70] Dimaz Ankaa Wijaya, Joseph Liu, Ron Steinfeld, Dongxi Liu, and Tsz Hon Yuen. Anonymity reduction attacks to monero. In Fuchun Guo, Xinyi Huang, and Moti Yung, editors, *Information Security and Cryptology*, pages 86–100, Cham, 2019. Springer International Publishing.

[71] Jeffrey Wilke. go-ethereum, December 2013. `https://github.com/ethereum/go-ethereum`, Last visit February 6 2022.

[72] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[73] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-Ning Dai. Xblock-eth: Extracting and exploring blockchain data from ethereum. *IEEE Open Journal of the Computer Society*, 1:95–106, 2020.

Abbreviations

ABI	Application Binary Interface
AE	Actual Events
BIP	Bitcoin Improvement Proposal
EE	Expected Events
ECDSA	Elliptic Curve Digital Signature Algorithm
EOA	Externally Owned Account
ERC	Ethereum Request for Comment
HDD	Hard Disk Drive
LMDB	Lightning Memory-Mapped Database
LSB	Least Significant Bit
P2M	Pay-to-multisig
P2PK	Pay-to-pubkey
P2PKH	Pay-to-pubkey-hash
P2SH	Pay-to-script-hash
P2WPKH	Pay-to-witness-pubkey-hash
P2WSH	Pay-to-witness-script-hash
P2TR	Pay-to-taproot
PoW	Proof of Work
OS	Operating System
segwit	Segregated Witness
SSD	Solid-State Drive
TXO	Transaction Output
UTXO	Unspent Transaction Output

List of Figures

List of Tables

Appendix A

Installation Guidelines

This project uses "pyenv" and "pipenv" for managing the python version, dependencies and even running commands. The recommendation of using these two tools in combination is from https://gioele.io/pyenv-pipenv. A python "pip" module of the package has not been published yet, but may be provided at a later date. Even so, this guideline serves more as an installation guide for a development environment.

Clone and build Monero LMDB

```
git clone https://github.com/monero-project/monero
cd monero/external/db_drivers/liblmdb
make
```

The following environment variables need to be set when installing the project's dependencies, in particular when installing Python LMDB:

```
export LMDB_FORCE_SYSTEM=1
export LMDB_INCLUDEDIR=~/monero/external/db_drivers/liblmdb
export LMDB_LIBDIR=~/monero/external/db_drivers/liblmdb
```

If in doubt, consult this Monero Stackexchange answer https://monero.stackexchange.com/questions/12234/python-lmdb-version-mismatch.

Prepare monero-serialize

`monero-serialize` requires local, unpublished patches. For this clone the repository and checkout the patched branch:

```
git clone https://github.com/TheCharlatan/monero-serialize
cd ~/monero-serialize
git checkout txMetaData
```

Install pyenv

On Ubuntu/Debian:

```
sudo apt-get update; sudo apt-get install make build-essential libssl-
    dev zlib1g-dev \
libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm \
libncursesw5-dev xz-utils tk-dev libxml2-dev libxmlsec1-dev libffi-dev
    liblzma-dev
```

Then for ease of use:

```
curl https://pyenv.run | bash
```

Read the pyenv installation manual at https://github.com/pyenv/pyenv on how to configure the correct environment variables for pyenv.

Install pipenv

```
pyenv global 3.7.0
pip install pipenv
```

Clone the project

```
git clone https://github.com/TheCharlatan/blockchain-parser
```

Install dependencies with pipenv

```
pipenv install ~/monero-serialize
```

Running the Script

The Monero LMDB parser requires the following path to be set to the LMDB library:

```
LD_LIBRARY_PATH="/usr/local/lib:$USER/monero/external/db_drivers/liblmdb
    " \
     pipenv run python main.py --help
```

The help text should self-describe the usage of the parsers. The parsers read data directly from the blockchain database. The blockchain-parser thus requires access to the directory where the blockchain database files are located.

IDE Integration

The python path that pipenv is configured to after installation is made available with:

```
pipenv --py
```

Appendix B

Contents of the ZIP Archive

1. This thesis as PDF
2. This thesis as LaTeX source archived in a `zip` file, including the figures
3. Figures in their .drawio format in a directory called `figures_drawio`
4. The blockchain-parser source code in a directory called `blockchain-parser`
5. Midterm presentation slides as a `PDF` document
6. The histogram datasets from the evaluation as `CSV` files, in a directory called `datasets`
7. The extracted strings discussed in the thesis as `TXT` files, in a directory called `extracted_strings`